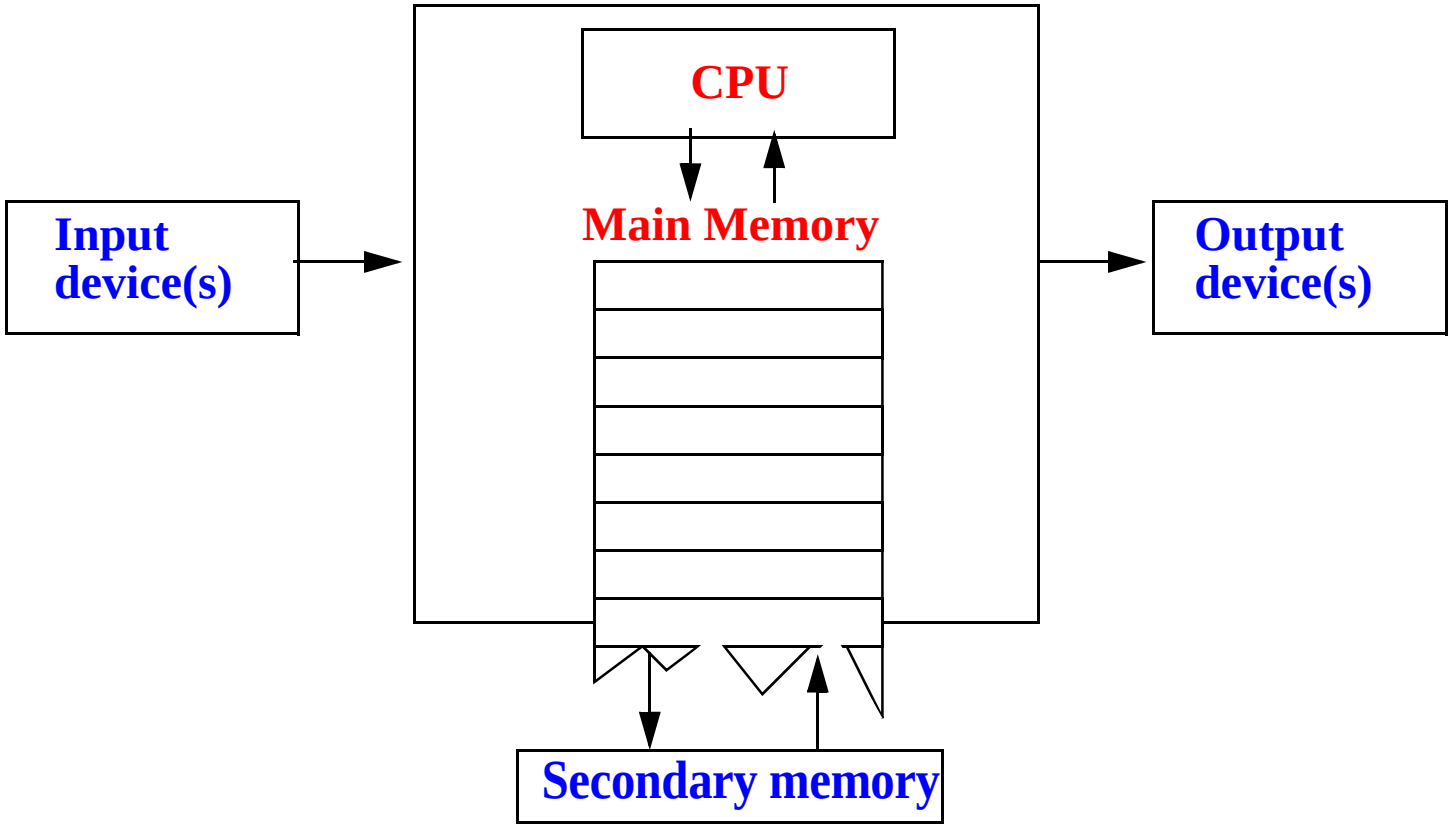
CSE 11: Lecture 3

- Numeric operators and the String concatenation operator
- The assignment operator and assignment statements
- Programming and logic
- Comparison operators and boolean operators
- Conditional statements
- Iteration
- Methods: parameters and return values

(Reading: Savitch, parts of Ch. 2 , 3, and 4)

Programming and operators

- Recall the basic organization of a digital computer:



- Values of variables reside in main memory (we talked about those last time)
- Operators take those values into the CPU and do interesting things with them (we'll talk about them now)

Operators: computing with data

- An operator operates on data (variables or constants or the results of other operations)
  - An operator *returns a value* that can be used in further computation
  - An operator may also have "*side effects*" (things it does other than returning a value): it may change the value of variables, print something, etc.

- The pieces of data an operator operates on are its *arguments* or *operands*
  - If an operator takes one argument it's called "unary"; a "binary" operator takes two, a "ternary" operator takes three

- Operators are used to write expressions:
  - an expression is a sequence of data and operators that take the data and perform computations to produce a value (and side effects...)
- Operators have a precedence and associativity direction that determines the order of application of operators in an expression
  - The associativity and precedence is familiar from arithmetic and algebra
  - Associativity and precedence can be empasized, or overridden, by use of parentheses **( )**

Numerical operators in Java

- To start with, we'll consider these binary numeric operators (they operate on numeric and char values):

`+`	plus	(this is also the String concatenation operator)
`-`	minus	
`*`	times	
`/`	divide	
`%`	remainder	

Addition and subtraction operators: +,-

- `foo - bar + baz` means

  `(foo - bar) + baz` , which in general is not the same as `foo - (bar + baz)`

- `-` subtracts the second argument from the first, and returns the result.
  It has no side effects.

- `+` adds two numeric arguments, and returns the result.
  It has no side effects.

The String concatenation operator +

- If at least one argument to the binary operator **+** is a String object, it acts as a string concatenation operator (not a numeric operator):
  - If the other argument is not a String, it is converted to one
  - The result of the concatenation is a String object

```
String day = "Friday";
String wish = "You wish";
System.out.println(wish + " today were " + day);
//  prints:    You wish today were Friday

String output = "The answer is:" + 42;
// now output is a String with value  "The answer is:42"
System.out.println(output);
// prints:     The answer is:42
```

Multiplication, division, remainder operators: *, /, %

- `foo / bar * baz % boz` same as `((foo/bar)*baz)% boz`
  `foo + bar * baz - boz` same as `(foo+(bar*baz))- boz`

- `*` multiplies two numeric arguments, and returns the result.
  It has no side effects.

- `/` divides two numeric arguments, and returns the result.
  It has no side effects.

- On integer arguments, `/` returns the integer part of the quotient:
  for example, `13 / 5` has value `2`

- `%` returns the remainder after division:
  for example, `13 % 5` has value `3` (also called “modulus” operator)

Arithmetic expressions in Java

Using Java operators, convert these mathematical formulas to Java expressions:

$b^2 - 4ac$	`b * b - 4 * a * c`
$x(y + z)$	`x * (y + z)`
$\frac{1}{x^3 + x + 3}$	`1 / (x * x * x + x + 3)`
$\frac{a + b}{c - d}$	`(a + b) / (c - d)`

The assignment operator = and assignment statements

- The assignment operator `=` is used to write assignment statements

- Assignment statements have this form (syntax):
  `variable = expression ;`
  ... and they have this semantics:
  - First, compute the value of the expression on the right-hand side
  - Then, make the variable on the left-hand side have that value (a “side effect”!)

- The variable must have been declared earlier in the program (of course)

- The expression can be any expression that has a type compatible with the type of the variable (we’ll talk about type compatibility later)

- Assignment is very different from equality!
  - Read the statement `a=b;` as "a gets assigned the value of b", NOT "a equals b"!
  - For example: the statement `a=a+1;` makes sense, but “a equals a plus one” does not

Programming and logic

- Computers are universal logic machines (though they can be programmed to act illogically!)
- To get them to do what you want them to do (the goal of programming), it really helps if you understand logic
- One important aspect of this is the syntax and semantics of Boolean expressions
- In general, Boolean expressions are expressions which have a value that is interpreted as being either *True* or *False*
- In Java, Boolean expressions have a value of type **`boolean`**, which is not a numeric type, and cannot be cast to or from any numeric type
  - a Java expression of **`boolean`** type has one of two possible values, which can be expressed with the boolean literal constants **`true`** or **`false`**
- We will look at comparison operators which generate boolean values, and boolean operators which combine boolean values, and at using boolean expressions to control branching and iteration

Comparison operators in Java

- Comparison operators take two expressions as arguments, and return the boolean value **`true`** or **`false`** depending on whether the comparison of the values of the expressions is true or false:

```
a == b   // has value true if a equals b;  else false

a != b   // has value false if a equals b;  else true

a < b    // has value true if a is less than b; else false

a > b    // has value true if a is greater than b; else false

a <= b   // has value true if a is less than or equal to b;
         // else false

a >= b   // has value true if a is greater than or equal to b;
         // else false
```

- These operators have no side effects

Boolean operators in Java

- Logical (boolean) operators take boolean-valued arguments, and return boolean values

```
a && b  // value true if both a and b are true, else false
        // (logical conjunction)

a || b  // value true if either a or b or both are true, else false
        // (logical disjunction... "inclusive" or)

!a      // value true if a is false, else false
        // (logical negation)
```

- These operators take as argument *only* boolean type operands
- These operators have no side effects

Semantics of boolean operators in general

- The logical semantics of the boolean operators can be summarized in a "truth table":

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

- From the properties of the operators, you can deduce the useful DeMorgan's laws:

  `!(A && B)` is equivalent to `!A || !B`
  `!(A || B)` is equivalent to `!A && !B`

- ... and the double-negation law:
  `!! A` is equivalent to `A`

Practice with Boolean expressions

- Assuming the following declarations...

```
int num = 3;
double x = 4.0;
char ch = 'Z';
```

- ... what are the values of these expressions?

```
3 == 3                                          // true
num == 3                                        // true
x > num                                         // true
x > num > 2                                     // illegal!
x > num   &&   num > 2                          // true
!(x > num)  &&  num == 3                        // false
!(x > num)  ||  num == 3                        // true
x < num  ||  num >= 3 + 8  ||  ch == 'A'        // false
true && true                                    // true
true && false                                   // false
false && false                                  // false
true || true                                    // true
false || true                                   // true
false || false                                  // false
```

Kinds of statements in Java

- We have seen declaration statements, like

```
char c;
Timer my_timer;
double lower = 0.0, upper = 1e99;
final int FIFTY = 50;
int year;
```

- ... and expression statements, i.e. an expression followed by a semicolon "`;`" like

```
c = 'c';
lower++;
System.out.println(lower + FIFTY);
year = 2001;
```

- ... and compound statements (code blocks), i.e. a sequence of 0 or more statements enclosed in curly braces "`{}`"

```
{
  <statement>
  <statement>
...
  <statement>
}
```

- Next we will look at conditional statements and iterative statements

Conditional control (branching) statements in Java

- The statements in a progam are executed in order when the program runs...

```
public static void main(String args[]) {
   <statement_1>
   <statement_2>
    ...
   <statement_n>
}
```

- *Conditional control* statements contain substatements which may or may not be executed, depending on the value of a Boolean expression

- Conditional control of statement execution is an essential part of programming a computer to do something interesting!

- Conditional control constructs in Java:
  if statements
  if-else statements
  switch statements (later)

The if statement

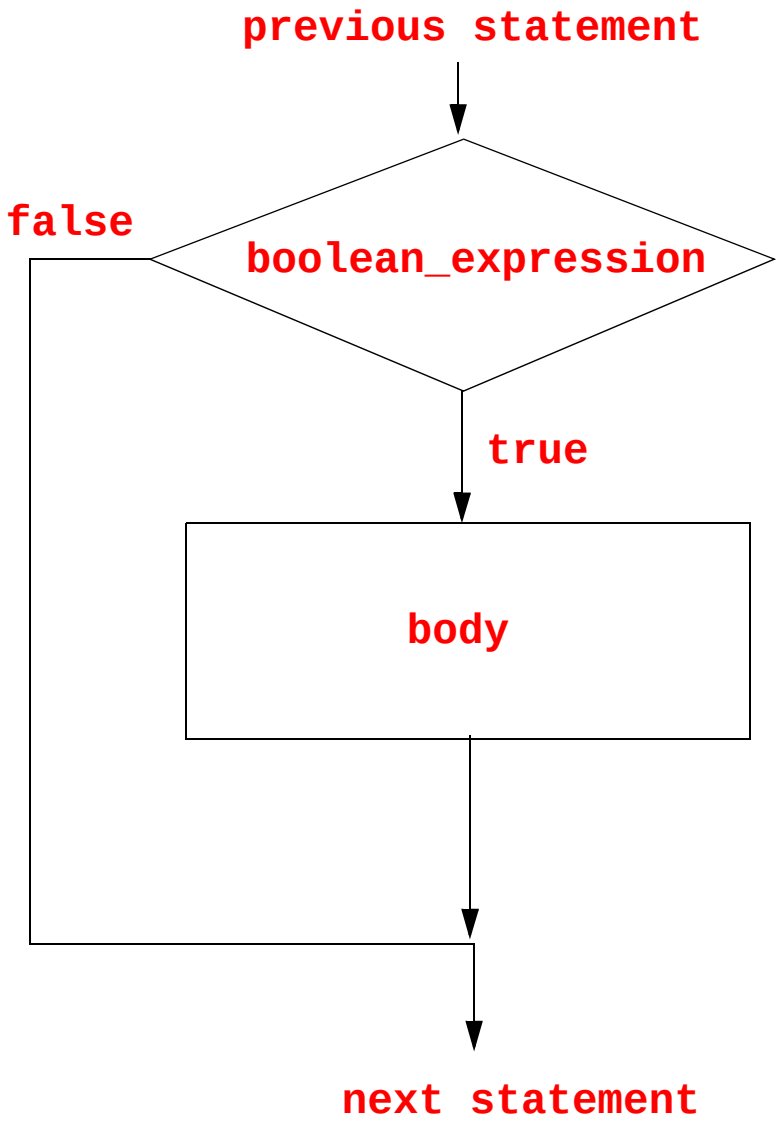
- Syntax: an **`if`** statement has the form

  `if (<expression>) <statement>`

- Semantics: when an **`if`** statement is executed...
  - the expression **`<expression>`** is evaluated
  - if its value is boolean true, the body **`<statement>`** is executed
  - if its value is boolean false, the body **`<statement>`** is not executed

- Note that the if statement body **`<statement>`** can be any kind of statement: simple, compound, or even a conditional one!

If statement flow chart



Practice with the if statement

- What (if anything) is printed by the following statements?

```
if (3>4)  System.out.print("Hi there\n");


if (false)    {
  System.out.print("Hi ");
  System.out.println("there");
}


int num=99;


if(num < 100)
  if(num > 0)
    if(num != 33)
      System.out.println("okay");
```

The if-else statement

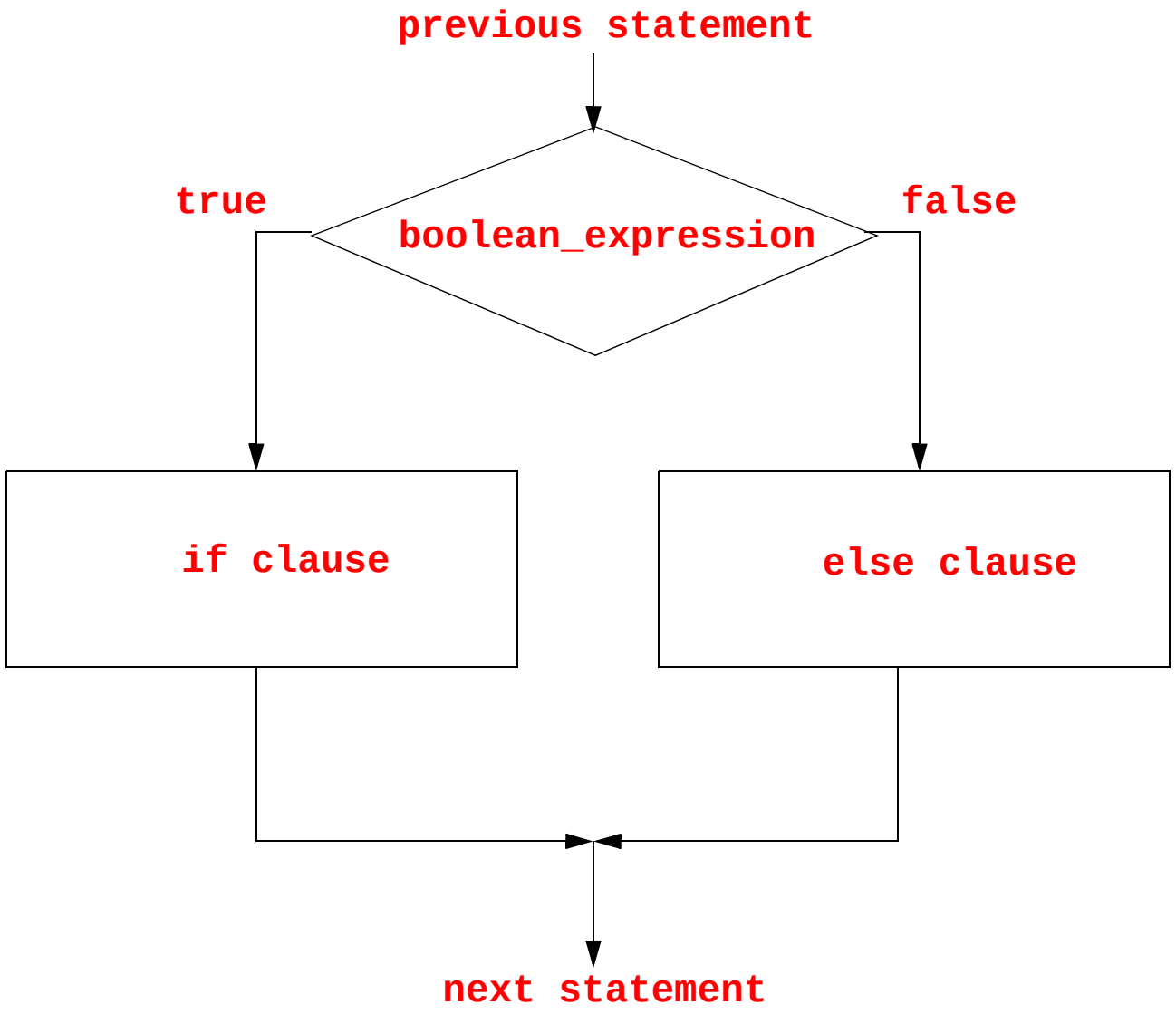
- Syntax: an **if-else** statement has the form

  if (<expression>) <statement1> else <statement2>

- Semantics: when an **if-else** statement is executed...
  - the expression **<expression>** is evaluated
  - if its value is true, the if-clause **<statement1>** is executed
  - if its value is false, the else-clause **<statement2>** is executed

- Note that **<statement1>, <statement2>** can be any kind of statements, even compound or conditional ones!

If-else statement flow chart



Practice with the if-else statement

- What (if anything) is printed by the following statements?

```
if (3>4)  System.out.print("Hi there\n");
else  System.out.print("Bye there\n");

int num=99;

if (num < 100)
   System.out.print("num is less than 100\n");
else if (num > 50)
   System.out.print("num is bigger than 50\n");


if (num > 200)
    System.out.print("num is more than 200\n");
else if (num > 100)
    System.out.print("num is between 101 and 200\n");
  else if (num > 50)
      System.out.print("num is between 51 and 100\n");
    else if (num > 0)
        System.out.print("num is between 1 and 50\n");
```

Iterative (looping) control statements in Java

- Iterative control statements contain substatements which are executed iteratively (i.e., over and over)

- Iteration is another essential part of programming a computer to do something interesting

  - In fact, with iteration, a program can compute *any* function...
  - Iteration is an extremely powerful computational trick, and so can be extremely tricky to get right

- Iterative control constructs in Java:
  while statements
  do-while statements
  for statements (later)

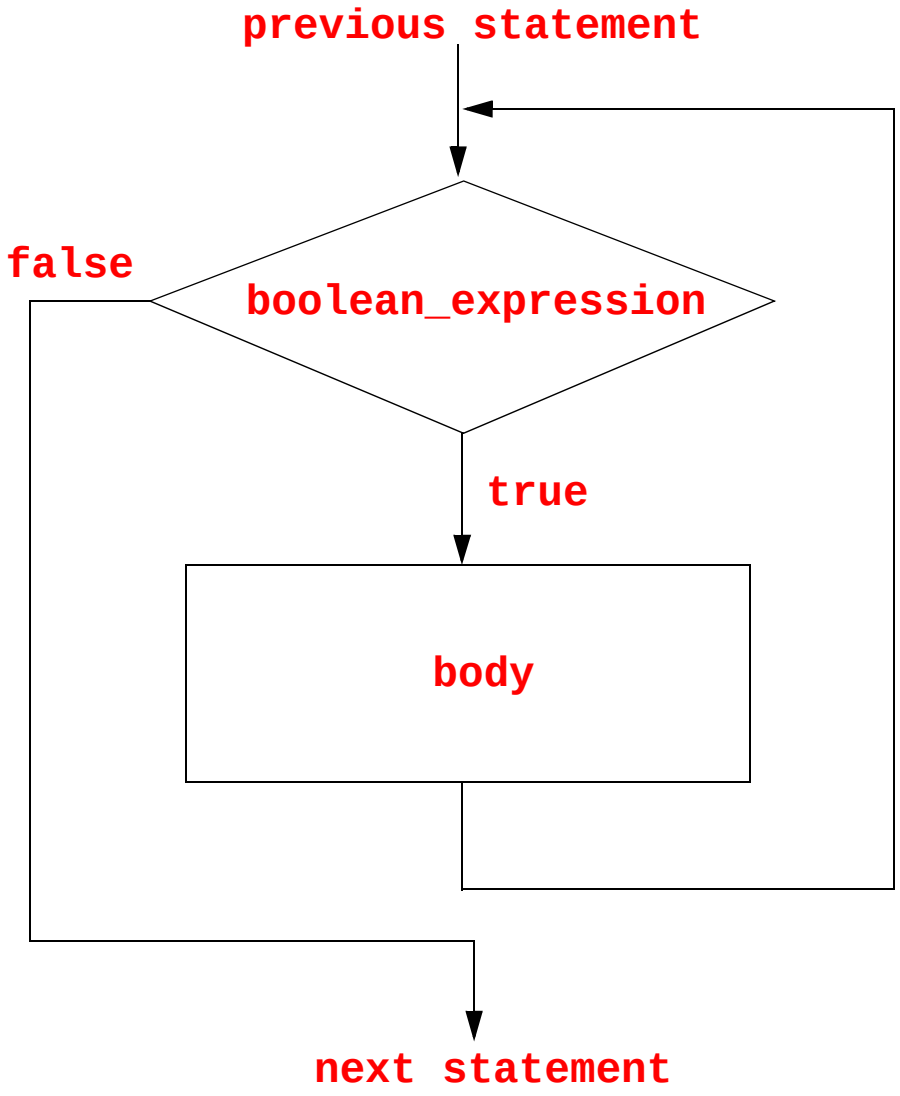
The while statement

- Syntax: a while statement in Java has the form

  `while (<boolean_expression>) <statement>`

- Semantics: when a while statement is executed...
  - The boolean expression is evaluated.
    - If it is “true”, the loop body **`<statement>`** is executed, and then the while statement is executed again
    - If it is “false”, the loop body **`<statement>`** is not executed, and control passes beyond the while statement
- The loop body can be any kind of Java statement

While-loop flowchart



The advantage of iteration...

- Compute the average of 100 integers entered from the terminal

```
int n, sum=0;

System.out.print("Enter next number: ");
n = Savitch.readLineInt();
sum = sum + n;
System.out.print("Enter next number: ");
n = Savitch.readLineInt();
sum = sum + n;
...           // 100 of these!

System.out.println("The average is " + sum / 100.0 );
```

- This takes a lot of lines of code! You can write it much more compactly using an iterative control construct

- (Any time you find yourself writing repeated code segments like this, it is a sign you should think about writing a loop instead.)

Example: averaging 100 integers

- Easy with iteration:

```
int n, sum=0, i=0;

while(i < 100)  {
   System.out.print("Enter a number: ");
   n = SavitchIn.readLineInt();
   sum = sum + n;
   i = i+1;
}

System.out.println("The average is " + sum / 100.0 );
```

Example: averaging integers until a negative one is entered

- Easy with iteration, impossible without it (or its computational equivalent):

```
long n, sum=0, nentered=0;

System.out.print("Enter a number: ");
n = SavitchIn.readLineInt();

while(n >= 0) {
   sum = sum + n;
   nentered = nentered + 1;
   System.out.print("Enter a number: ");
   n = SavitchIn.readLineInt();
}

if(nentered > 0)
  System.out.println("The average is " + sum / (double)nentered);
```

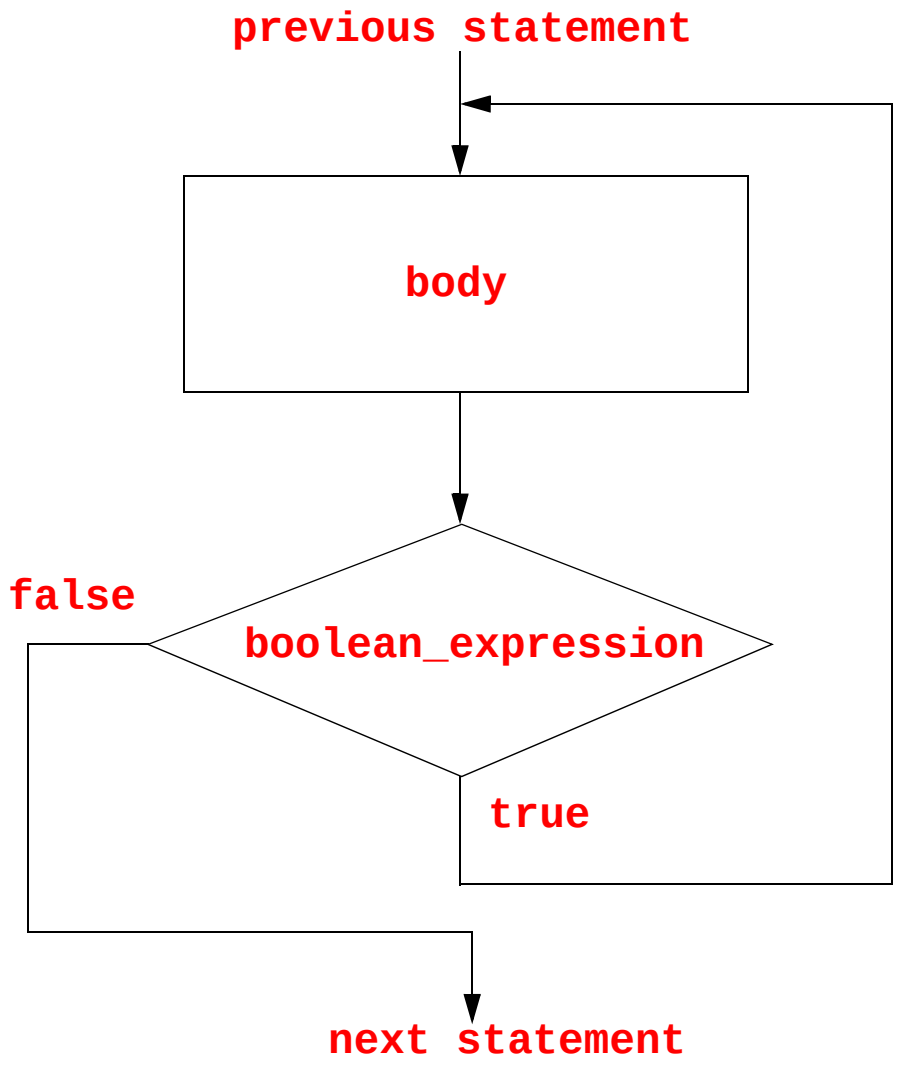
The do-while statement

- Syntax: a do-while statement has the form

```
do <statement> while (<boolean_expression>);
```

- Semantics: when a do-while statement is executed...
  - the loop body **`<statement>`** is executed.
  - The boolean expression is evaluated.
    - If it is “true”, the do-while statement is executed again
    - If it is “false”, execution passes to the statement after the **`do-while`**
- The loop body can be any kind of Java statement

Do-while loop flowchart



Example: averaging 100 integers

```
long n, sum=0, i=0;

do  {
   System.out.print("Enter a number: ");
   n = SavitchIn.readLineInt();
   sum = sum + n;
   i = i+1;
} while(i < 100);

System.out.println("The average is " + sum / 100.);
```

Example: averaging integers until a negative one is entered

```
int n, sum=0, nentered=0;

do  {
   System.out.print("Enter a number: ");
   n = SavitchIn.readLineInt();
   if(n >= 0)
    {
       sum = sum + n;
       nentered = nentered + 1;
    }
} while(n >= 0);

if(nentered>0)
   System.out.println("The average is " + sum / (double)nentered);
```

Loops and methods are labor saving devices

- If you find yourself writing essentially the same code segment repeatedly in sequence, think about putting it in the body of a loop

- If you find yourself writing (or cutting and pasting) essentially the same code segment repeatedly various places in your program, think about putting it in the body of a *method*, which is our next topic

Levels of software structure

- At the lowest level, software is just bits in computer memory...
- ... but it is way too complicated for our puny brains to think of it that way
- Instead, in a high-level language, you have simpler and more intuitive ways to organize how you talk to the machine:
  - Variables and literal constants are convenient ways to refer to patterns of bits
  - Expressions using operators are nice ways to specify operations on those patterns of bits
  - Simple statements are built from expressions in a straightforward way
  - Conditional branching and iteration give simple but powerful ways of organizing statements
  - Methods permit grouping statements together, and referring to the group by name
  - Classes and objects group methods and variables together
  - Packages group classes together
- We have talked about some of these levels already... methods next; classes, objects, packages to come

Methods

- A method (or function, or procedure) is a sequence of statements that has...
  - a way to refer to it
    - its *name* (an identifier)
  - a way to give information to it
    - a specification of the number, ordering, and types of its *arguments*
  - a specification of its return value type (or **`void`** if none)
- A method's sequence of statements form a compound statement, called the *body* of the method; these statements are executed when the method is called
- The method body may compute a return value (or not), and may perform side effects (or not)
- So, methods are similar to operators... except you can define your own methods!

Defining methods in Java

- In Java, every method definition must be inside a class definition
  - ... you can't have a method definition all by itself in a file

- You have already been defining Java methods in your programming assignments. For example:

```
public class FourDigits2 {

   public static void main (String args[]) {
      System.out.print("Enter a 4-digit number: ");
      int num = SavitchIn.readLineInt();
      // etc...
   }
}
```

- `main` here is a "public static void" method, defined inside the `FourDigits2` class.

  Lets look at it more closely...

The syntax of method definitions

- A method definition has two important parts:
  - the heading: usually written on one line

```
public static void main (String args[])
```

    - the heading (also called the “prototype” or “signature” or “header”) specifies the name of the function, number and type of arguments, return type, and some other properties

  - the body: a compound statement (enclosed in “curly braces” **{}**) that comes right after the heading

```
{
   System.out.print("Enter a 4-digit number: ");
   int num = SavitchIn.readLineInt();
   // etc...
}
```

    - The body specifies the sequence of statements that are executed when the method is invoked (called)

Static and instance methods

- There are two kinds of methods that can appear inside a Java class:
  - static methods
  - instance methods
- If you use the keyword **`static`** in the method header (just before the return type), it is a static method; if you do not use that keyword, it is an instance method
- A static method is associated with the *class* within which it is defined
  - You do not need to create an object that is an instance of the class to use (invoke or call) a static method
  - Static methods are also called “class methods”
- An instance method is associated with *each object* that is an instance of the class within which it is defined
  - You (or someone) must create an object that is an instance of the class to use (invoke or call) an instance method
- We will concentrate on static methods first (objects not needed!)

Defining static methods

- A header of a static method has the form

```
<visibility> static <typename> <identifier> ( <argumentlist> )
```

  - **<visibility>** is either **public**, **private**, **protected**, or (if missing) package; we will consider only public visibility for now
  - **<typename>** specifies the datatype of value the method returns (or **void**, if none)
  - **<identifier>** specifies the name of the method
  - **<argumentlist>** is the formal argument or parameter list that specifies the number, type, and ordering of the method's arguments

Static method headers: examples

```
public static void greet()
```

... for a private static method named **`greet`**, which takes no arguments and returns nothing (but probably has side effects)

```
public static void setValues(int ival, double dval)
```

... for a public static method named **`SetValue`**, which takes one integer argument and one double argument, and returns nothing (but probably has side effects)

```
public static int add3Values(int what, int why, int where)
```

... for a method named **`add3Values`** that takes three int arguments, and returns an int

```
public static int add3Values(int what, why, where)  // wroong!
```

... NO! every argument must have its type individually specified...

Calling methods: arguments and return values

- The definition of a method has a formal argument list which specifies the pieces of information you must pass to the method when you call it
  - The call of the method will have an actual argument list which specifies the information you are passing to the method
  - The actual argument list must match the formal argument list in *number* and *type* of arguments
  - The identifiers in the formal argument list are assigned the corresponding values in the actual argument list when the method is called, and these identifiers act like variable names that can be used in the body of the method
- A method call is a kind of expression
  - the value of this expression is the value returned from the method, if any
  - ... so calls to void methods are examples of expressions without a value
- A method may also have side effects!...

Return statements in methods

- A return statement has the form
  `return <expression>;`
  or
  `return;`

- Every method that is declared to return a value (not void) must contain at least one return statement of the first form
  - when a return statement is executed, the value of the expression is computed
  - execution of the method stops, and the method returns the value of the expression as the value of the method call expression
  - program execution resumes at the point of the method call

- A method that is declared to return void may contain a return statement; if so it must be of the second form
  - when a return statement is executed in a void method, the method just returns (it stops executing, and program execution resumes at the point of call)
  - this also happens after the last statement in the body is executed, so a return statement is not necessary in a void method

Arguments and values: an example

- For example: consider this static method definition, in the public class **Foo**:

```
public class Foo {

   public static double crunch(int x, double obj, char ch) {
      if (ch == 'x') {
         System.out.println("Booyah");
         return 0.0;
      } else
         while (x>0) {
            obj = obj * obj;
            x = x - 1;
         }

      return obj;
   }
}
```

- Note that this method has 3 formal parameters, so it must be called with 3 actual arguments:
  the first one should be of type **int**, the second of type **double**, the third of type **char**
  (or a type that Java can automatically convert to these according to the rules for **=**)

Arguments and values: an example, cont'd

- Consider this code, inside the body of some other method:

First, declare and initialize some variables...

```
int x=3, y=4;
double baz, bar=3.14;
char c='?';
double result;
```

Note that this call to **`crunch`** *does not use the return value; it is only useful for its side-effects. What does this call to* **`crunch`** *do?*

```
crunch(x,bar,'x');
```

If you are calling a method from inside another class, you will have to specify the name of the class that defines the method, like this:

```
Foo.crunch(x,bar,'x');
```

Here is a call to **`crunch`** *where the return value is used. What value is assigned to* **`result`**?

```
result = crunch(99/x - 2, 1.0, c) * 22.2;
```

What exactly happens when a method is called?

- First, the *values* of the actual arguments are obtained

```
crunch(    99/x - 2,     1.0,          c   )

             31          1.0        '?'
```

- Next, these values are assigned to the corresponding formal arguments in the method definition header

```
                                    31          1.0      '?'

public static double  crunch(int x, double obj, char ch)
{
    //   and the formal arguments with these values can be used here in
    //   the function body just like variables
}
```

- Finally, the statements in the body of the method definition are executed in order, until a `return` statement specifies what value to return from the method (if any!)

Next time

- ✔ Comments in code and coding style
- ✔ Procedural and object-oriented programming
- ✔ Designing classes for object-oriented programming
- ✔ Abstract Data Types (ADT's)
- ✔ Instance variables and instance methods
- ✔ Variable and method visibility
- ✔ Constructors
- ✔ The "this" variable
- ✔ Variables of reference type vs. variables of primitive type

(Reading: Savitch, part of Ch. 2, Ch. 4, and Ch. 5)