Conditional Statements

- Allows the execution of code if some boolean value holds true
  - Examples of boolean experessions: true, false, i < 10, s.equals("Hi");
  - Basically anything that evaluates to true or false.
- If statement

```
if (boolean_expression) {
      statements
} else {
      statements
}
```

- If there is only one statement per block, you can remove the "{ }".
- For readability, consider always using "{ }".

```
if (boolean_expression)
      statement
else
      statement
```

- Java allows "null bodies."
- Generally bad practice since condition statements can be written where null bodies are not needed.

```
if (boolean_expression)
      ; // or { }
else {
      statements
}
```

Booleans

- Specifically true and false values.
- **true** and **false** are keywords in Java.
- Not the same as 0 and 1 like other languages (compiler will complain)

Relationals

- Examples: ==, !=, <, <=, >, >=
- Left to right precedence.
- Can change order of operations using ( )
- Recall: == for objects refer to the same object (i.e. same memory location).

Example

```
Object x = new Object(); // just a plain ol' object.
Object y = x;
if (x == y) {
      System.out.println("x == y"); // y refers to x
} else {
      System.out.println("x != y");
}
```

Logical Operators

- !, && , ||
  - short-circuit: && - right expression is evaluated only if left expression is true
  - short-circuit: || - right expression is evaluated only if left expression is false
- Change order using "( )"
- Note: Java has logical bit operators '&' and '|' . These are not the same as boolean expressions, but actually perform AND / OR on the bits.

Example

```
int z = 5;

if (!((z – 3) > 0)) {
      System.out.println("true");
} else {
      System.out.println("false"); // Prints False
}
```

Avoid defining variables within conditions

```
int a = 0, b = 0, c = 0, d = 0;
if (a++ > 0) // bad style, hard to read
if ((d = b + c) <= 0) // legal, bad style, hard to read
```

Simplify boolean logic

```
if (a < b)
      return true;
else
      return false;
////
return  a < b; // Does the same thing!
```

Statements after returns

```
if (a < b) {
      System.out.println("a<b");
      return a;
} else {
      System.out.println("a>=b");
      return b;
}
System.out.println("not exec");// Compile error, code is unreachable
```

Scopes

- Variables can be defined within a conditional scope.

```
if (a >= 0) {
      int i = 3;
      i++;
      System.out.println(i);
}
System.out.println(i); // ERROR: i is valid within the if block
```

- You cannot declare a variable with the same name as something already in scope.

```
int i = 0;
if (a > 4) {
      int i = 3; // illegal!
      i++;
      System.out.println(i);
}
```

- i can be re-declared once it goes out of scope.

```
if (a > 4) {
      int i = 3;
      i++;
      System.out.println(i)
}
int i = 7; // OK
System.out.println(i);
```

While Loop

```
while (boolean_expression) {
      statements;
}
```

- Statements can be executed anywhere from 0 to infinite times.
- Boolean expression is checked at the start of the loop.
    - Executes statements if the boolean expression evaluates to **true**.
- To guarantee executing something at least once, that code will have to exist before while loop. (redundant code).
- Similar to **if** statements, "{ }" required for more than one statement.

Example

```
int i = 0;
while(i < 10) {
      i++;
      System.out.println(i);
} // Prints 1 -> 10
```

Do Loop

```
do {
      statements;
} while (boolean expression);
```

- Statements can be executed anywhere from 1 to infinite times.
- Boolean expression is checked at the end of the first (and subsequent) iterations.
    - Executes statements if the boolean expression evaluates to **true**.

Example

```
int i = 0;
do {
      i++;
      System.out.println(i);
} while(i < 10); // Prints 1 -> 10
```

Use ++ and – Properly

- For readability purposes, it's generally a bad idea to change variables in boolean expressions.

Example (of what not to do)

```
int n = 1;
do {
   System.out.println(n);
} while (n++ <=3); // Outputs 1 2 3 4

////

int n = 1;
do {
   System.out.println(n);
} while (++n <=3); // Outputs 1 2 3
```

For Loop

```
for (initial action; boolean expression; update action) {
   statements;
}
```

- Statements can be executed anywhere from 0 to infinite times.
- initial action is executed first (regardless if boolean expression evaluates to true or not.
- Boolean expression is checked before each iteration of the loop.
    - Executes statements if the boolean expression evaluates to true.
- Update action is executed after the statements are executed.
    - Usually does something to change initial assignment.
- Boolean expression is checked after update action is performed.
- Note: Any while loop can be written as a for loop (and vice versa).

Examples

```
for (int i = 0; i < 10; i++) {
      System.out.println(i); // prints 0 – 9
}

int i = 0;
while (i < 10) {
      System.out.println(i); // prints 0 - 9
      i++;
}
```

```
i = 0;
for (; i < 10;) { // null initial and update statements ☹
      System.out.println(i); // prints 0 - 9
      i++;
}

for (int j = 0;;j++) { // null boolean expression statement ☹
      System.out.println(j * j); // overflow
}
```

Scoping with Loops

- Similar to scoping with conditional statements, variables declared within "{ }" are only valid in that block.

Examples

```
int sum = 0;
for (int i = 1; i <= 10; i++)
{
   sum = sum + i;
}
System.out.println(i);              // ERROR: no i here!

int i = 3;
int sum = 0;
for (int i = 1; i <= 10; i++)       // ERROR: i is already declared
{  }

int sum = 0;
do {
   int count = i + 1;
} while (count != 999);             // ERROR: no count here!
```

Breaks and Continue

- **break –** breaks out of the current loop.
- **continue** – breaks out of the current iteration of the loop and performs the next iteration.

Example

```
int i = 0;
while (true) {
      i++;
      System.out.println(i);

      if (i < 100) {
            continue;
      } else {
            break;
      }
}
System.out.println("Outta the loop");
// 1 -> 100
// Outta the loop
```

Recursion

- A recursive method is a method that makes a call to itself.
- Useful for problems that can be solved by using the results of similar subproblems.

Example: Factorial

- N! = N * (N -1) * (N – 2) * … * 2 * 1
- N! = N * (N – 1)!
- (N – 1)! = (N – 1) * (N – 2)!
- …
- Note: 0! = 1, 1! = 1

```
public int factorial (int n) {
      if (n == 0 || n == 1) // base case
            return 1;

      return n * factorial(n-1);
}
```

Evaluation of factorial(4):
factorial(4) = 4 * factorial(3)
3 * factorial(2)
2 * factorial(1)
1
2 * 1
3 * 2
4 * 6
24

Example: Palindrome

- A palindrome is a String that is the same forwards and backwards.
- Examples: HANNAH, ABA, RACECAR, CIVIC, …
- We can take a String and checks the first / last characters and see if they're equal.
    - Continue checking the substring excluding the first and last characters that were just evaluated.

```
public static boolean isPalindrome(String s) {
      if (s.length() == 0 || s.length() == 1) // base case
            return true;

      if (s.charAt(0) == s.charAt(s.length() – 1))
            return isPalindrome(s.substring(1, s.length() – 1));

      return false;
}
```

- Any (primitive) recursive solution can be written in an iterative way.
- Any iterative solution can be written in a recursive way.

Call Stack

- Space allocated for each pending method in the method call chain.
- Keeps track of the current state for each level in the call stack.

Stack Overflow

- An event that happens when a program crashes due to running out of memory in its call stack.
- System designers avoid recursion since systems are designed to be as lean and efficient as possible.
- Others may prefer recursion for readability purposes (could result in a lot less code).

Infinite Recursion

- Similar to an infinite loop, except the function calls itself forever.

Example

```
// Forgetting to include the base case in the factorial method.
public int factorial (int n) {
      return n * factorial(n-1); // calls itself until stack overflow
}
```

Arrays

- You can allocate memory to store items of the same type with arrays.
- Recall, arrays are indexed from 0 to n – 1.
- The items in a Java array must be of the same type (unlike Python!)
- Arrays are objects (i.e. they're not primitive Java types).

Example

```
// declare array
int[] array;

// assign array of size 100 ints (i.e. 100 * 4 bytes)
array = new int[100];

// initialize elements
array[0] = 10;
array[1] = 20;

// extract elements
System.out.println(array[1]); // prints 20

// initialization shortcut
int[] array2 = {10, 20, 30}; // creates an array of size 3 ints

System.out.println(array2[2]); // prints 30

System.out.println(array[4]); // returns 0
System.out.println(array2[4]); // ERROR! ArrayIndexOutOfBounds
```

Array Object Fields

- **.length**
  - the number of elements allocated for the Array.
  - `int x = array.length; // not a method, but a value`
- **.clone()**
  - Makes a copy of the contents of the array.
  - `int[] array3 = array2.clone();`

Example

```
int[] array3 = array2.clone();
if (array3 == array2) {
	System.out.println("array3 == array2");
} else {
	System.out.println("array3 != array2"); // Two separate arrays
}

// loop through array (print out values).
for (int i = 0; i < array3.length; i++) {
	System.out.println(array3[i]);
} // 10, 20, 30
```