Revised: 3/4/2019

Java Logic and Repetition

Michael C. Hackett

Computer Science Department

Community College *of* Philadelphia

Lecture Topics

- Relational and Logical Operators
- Decision Structures
    - If/If-Else Structures
    - Switch Structures
    - Conditional Operator
- String Comparison

- Repetitive Structures
    - While Loops
    - Traditional For Loops
    - Do While Loops
- Branching Statements
- Infinite Loops
- Random Number Generators

Colors/Fonts

Local Variable Names	– Brown
Primitive data types	– Fuchsia
Literals	– Blue
Keywords	– Orange
Object names	– Green
Operators/Punctuation	– Black
Field Names	– Lt Blue
Method Names	– Purple
Parameter Names	– Gold
Comments	– Gray
Package Names	– Pink

Source Code	– **Consolas**
Output	– Courier New

Relational Operators

- Java’s relational operators are the same as Python’s.
  - Each operator returns true or false

==	Equality Operator
!=	Inequality Operator
>	Greater Than Operator
<	Less Than Operator
>=	Greater Than or Equal To Operator
<=	Less Than or Equal To Operator

Logical Operators

- Java's logical operators behave like Python's, but they are symbols instead of keywords.

&&	And Operator
||	Or Operator (Shift+\ on keyboard)
!	Not Operator

Operator Precedence

0. ( ) Expressions in parentheses are always evaluated first.
1. !, - Not Operator, Unary Negation (-5)
2. *, /, % Multiplication, Division, Modulus
3. +, - Addition, Subtraction
4. <, >, <=, >= Less than (or equal), Greater than (or equal)
5. ==, != Equal to, Not equal to
6. && And Operator
7. || Or Operator
8. =, +=, -=, *=, /=, %= Assignment and Augmented Assignment

If Statements

- An if statement in Java works like if statements in Python.
  - The code will be "skipped" if its Boolean expression evaluates to false.
- The syntax for an if statement in Java is shown below.
  - The biggest difference from Python is the inclusion of braces and the condition must be in parentheses.

```
if(Boolean Expression) {
  //code that will be
  //executed if the Boolean Expression
  //evaluates to true
}
```

If Statement

```java
int length = 50;
int maxLength = 100;

if(length >= 0 && length < maxLength) {
   System.out.print("This is a ");
   System.out.println("valid length.");
}
```

```
This is a valid length.
```

Else Clauses

- An else clause in Java works like an else clause in Python.
- The syntax for an else clause in Java is shown below.

```
if(Boolean Expression) {
  //code that will be
  //executed if the condition
  //evaluates to true
}
else {
  //code that will be
  //executed if the condition
  //evaluates to false
}
```

Else Clause

```
int day = 10;

if(day > 0 && day <= 30) {
   System.out.print("This is a valid ");
   System.out.println("day in September.");
}
else {
   System.out.print("This is not a valid ");
   System.out.println("day in September.");
}
```

```
This is a valid day in September.
```

Else Clause

```java
int day = 31;

if(day > 0 && day <= 30) {
   System.out.print("This is a valid ");
   System.out.println("day in September.");
}
else {
   System.out.print("This is not a valid ");
   System.out.println("day in September.");
}
```

```
This is not a valid day in September.
```

Else If Clauses

- An **else if** clause in Java works like an **elif** clause in Python.
  - There is no elif keyword in Java.
- The syntax for an else if clause in Java is shown below.

```
if(Boolean Expression 1) {
  //code that will be executed if the expression
  //evaluates to true
}
else if(Boolean Expression 2) {
  //code that will be executed if Boolean Expression 1 was false
  //and this Boolean Expression 2 evaluates to true
}
else {
  //code that will be executed if no previous expressions
  //evaluated to true
}
```

Else If Clauses

```
double temp = 215.5;

if(temp <= 32.0) {
   System.out.println("Water will freeze.");
}
else if(temp >= 212.0) {
   System.out.println("Water will boil.");
}
else {
   System.out.println("Water will be liquid.");
}
```

```
Water will boil.
```

If-Else Structure Rules

- The rules are the same as Python.

- If Statements
  - **Must** always be first.
  - May be followed by any number of else if clauses.
  - May be followed by one else clause.
- Else If Clauses
  - Optional.
  - **Must** follow an if statement or else if clause.
  - No limit to the number of else if clauses.
  - May be followed by one else clauses.
- Else Clauses
  - Optional.
  - **Must** follow an if statement or else if clause.
  - **Always** the last clause.

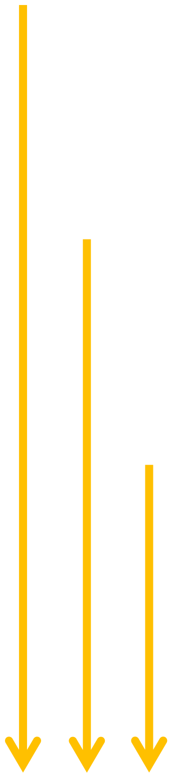
Variable Scope

- A variable’s ***scope*** determines where that variable can be used/accessed in a program.

- As we have seen, variables must be declared before the program can use them to store data.

- Such a variable’s scope will generally be the lines of code that follow its declaration.

Variable Scope



```
int value1 = 10;
System.out.print("Value 1 is: " + value1);

int value2 = 17;
System.out.print("Value 2 is: " + value2);

int sumOfValues = value1 + value2;
System.out.print("The sum is: " + sumOfValues);
```

Variable Scope

- A variable that is only accessible within a particular method, if statement, else clause, while loop, etc. has *local scope*.
  - Usually called a *local variable*.

- It is possible to have variables with *class scope*.
  - These variables are accessible anywhere within a class.
  - Usually called *global variables.*
  - We will see the use of different types of global variables when we move into object-oriented programming.

Variable Scope

```
public static void main(String[] args) {
    double totalAmount = 21.0;

    System.out.print("Total cost with shipping: $");
    if(totalAmount >= 30.0) {
        System.out.printf("%.2f", totalAmount);
    }
    else {
        double amountPlusShipping = totalAmount + 10.0;
        System.out.println("%.2f", amountPlusShipping);
    }
    System.out.print("Program Complete.");
}
```

Switch Structures

- A ***switch structure*** is a decision structure that allows multiple execution paths.
  - Unlike an if-else structure that only allows one execution path.
  - Python does not have a switch structure.

- Values used in a switch structure are limited to:
  - Primitive Types: int, char, byte, short
  - Object Types: String, Character, Byte, Short, Integer
  - Enumerated types (Not shown in this lecture.)

Switch Structures

- A switch structure contains (in braces):
  - Collections of statements that are each labeled by the keyword **case**.
  - A last, optional collection of statements that are labeled by the keyword **default**.

```
int myNumber = 2;

switch(myNumber) {
  case 0:  //ANY CODE HERE EXECUTES IF myNumber
           //WAS EQUAL TO 0
  case 1:  //ANY CODE HERE EXECUTES IF myNumber
           //WAS EQUAL TO 1
  default: //ANY CODE HERE IS EXECUTED IF NO
           //OTHER CASES WERE PREVIOUSLY MATCHED
}
```

Switch Structures

- The switch structure takes the value of the variable given to it, and tries to match it to a case.
  - It will execute that case's statements when it finds a match.
  - If a match was not found, and a default case is present, it will execute the default case's statements.

```
int myNumber = 4;
switch(myNumber) {
  case 5:   System.out.print("The number ");
            System.out.println("is five.");
  case 6:   System.out.print("The number ");
            System.out.println("is six.");
  default: System.out.print("This is ");
           System.out.println("the default.");
}
```

The value 4 doesn't match any of the cases, so the default case is executed.

```
This is the default.
```

Switch Structures

```java
int myNumber = 5;
switch(myNumber) {
   case 5:  System.out.print("The number ");
            System.out.println("is five.");
   case 6:  System.out.print("The number ");
            System.out.println("is six.");
   default: System.out.print("This is ");
            System.out.println("the default.");
}
```

The value 5 matched a case, so that case's code is executed.
But it doesn't stop there… it keeps going, executing all following cases.

```
The number is five.
The number is six.
This is the default.
```

Switch Structures

- This is the difference between if-else structures and switch structures.
  - In an if-else structure, there is only one execution path.
  - In a switch structure, there can be multiple execution paths.

```java
int myNumber = 6;
if(myNumber == 5) {
    System.out.print("The number ");
    System.out.println("is five.");
}
else if(myNumber == 6) {
    System.out.print("The number ");
    System.out.println("is six.");
}
else {
    System.out.print("This is ");
    System.out.println("the default.");
}
```

```java
int myNumber = 6;
switch(myNumber) {
  case 5:  System.out.print("The number ");
           System.out.println("is five.");
  case 6:  System.out.print("The number ");
           System.out.println("is six.");
  default: System.out.print("This is ");
           System.out.println("the default.");
}
```

Break Statements

- Using a break statement will prevent a case from falling through to the next case.

```
int myNumber = 5;
switch(myNumber) {
  case 5: System.out.print("The number ");
          System.out.println("is five.");
          break;
  case 6: System.out.print("The number ");
          System.out.println("is six.");
  default: System.out.print("This is ");
           System.out.println("the default.");
}
```

```
The number is five.
```

Break Statements

```
int myNumber = 5;
switch(myNumber) {
  case 5:  System.out.print("The number ");
           System.out.println("is five.");
  case 6:  System.out.print("The number ");
           System.out.println("is six.");
           break;
  default: System.out.print("This is ");
           System.out.println("the default.");
}
```

```
The number is five.
The number is six.
```

Switch Structures (Strings)

```
String moneyString = "USD";
switch(moneyString) {
  case "MXN": System.out.println("Peso");
              break;
  case "EUR": System.out.println("Euro");
              break;
  case "USD": System.out.println("US Dollar");
              break;
  default:    System.out.println("Unknown Currency");
}
```

```
US Dollar
```

Break Statements

- A switch structure with a break after every case has no benefit over an equivalent if-else structure.

```java
int myNumber = 5;
switch(myNumber) {
  case 5:  System.out.print("The number ");
           System.out.println("is five.");
           break;
  case 6:  System.out.print("The number ");
           System.out.println("is six.");
           break;
  default: System.out.print("This is ");
           System.out.println("the default.");
}
```

```java
int myNumber = 5;
if(myNumber == 5) {
    System.out.print("The number ");
    System.out.println("is five.");
}
else if(myNumber == 6) {
    System.out.print("The number ");
    System.out.println("is six.");
}
else {
    System.out.print("This is ");
    System.out.println("the default.");
}
```

In-line If Statements

- Also called “conditional operator” or “ternary operator”.
- Uses three operands.

Boolean Expression* ? *then do this* : *else do this

In-line If Statements

```
int fiveItems = 5;
int tenItems = 10;
boolean wantsTenItems = true;
int choice = 0;

choice = wantsTenItems ? tenItems : fiveItems;

System.out.println(choice);
```

```
10
```

In-line If Statements

```java
int fiveItems = 5;
int tenItems = 10;
boolean wantsTenItems = false;
int choice = 0;

choice = wantsTenItems ? tenItems : fiveItems;

System.out.println(choice);
```

```
5
```

In-line If Statements

```
int fiveItems = 5;
int tenItems = 10;
boolean wantsTenItems = false;
int choice = 0;

choice = wantsTenItems ? tenItems : fiveItems;

System.out.println(choice);
```

Equivalent to

```
if(wantsTenItems) {
    choice = tenItems;
}
else {
    choice = fiveItems;
}
```

In-line If Statements

```
int fiveItems = 5;
int tenItems = 10;
boolean wantsTenItems = false;
int choice = 0;

choice = wantsTenItems ? tenItems : fiveItems;
```

Removed

```
System.out.println(wantsTenItems ? tenItems : fiveItems);
```

Can be used in a method call to decide which to use as an argument

```
5
```

In-line If Statements

```
String stateName = "PENNSYLVANIA";
String stateAbbr = "PA";
boolean fullStateName = true;
String choice = "";

choice = fullStateName ? stateName.toLowerCase() : stateAbbr.toLowerCase();

System.out.println(choice);
```

```
pennsylvania
```

In-line If Statements

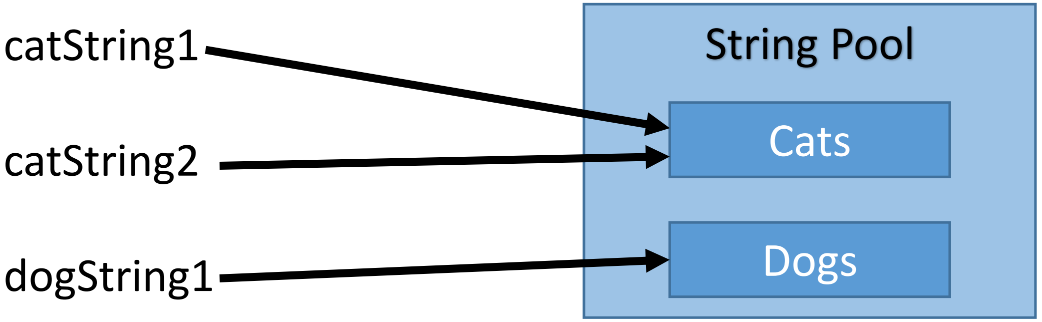
```java
String stateName = "PENNSYLVANIA";
String stateAbbr = "PA";
boolean fullStateName = false;
String choice = "";

choice = fullStateName ? stateName.toLowerCase() : stateAbbr.toLowerCase();

System.out.println(choice);
```

```
pa
```

String Pool

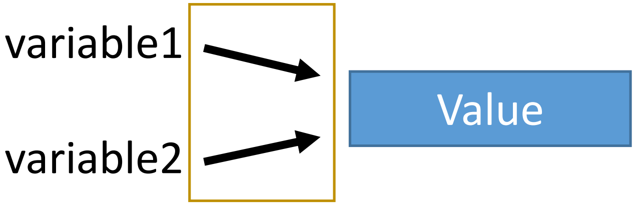
- In Java, when you assign a literal to a String variable, that value is added to a special section of memory called the String Pool.
- When you assign another String variable the same literal, it references the same value in the pool.
  - Java does this to help save space.

```
String catString1 = "Cats";
String catString2 = "Cats";
String dogString1 = "Dogs";
```

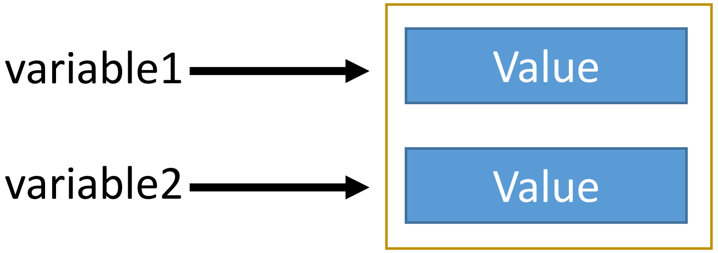


String Comparison

- There are two ways to test the equality of Strings:
  1. Testing if two String variables *reference* the same value in memory.



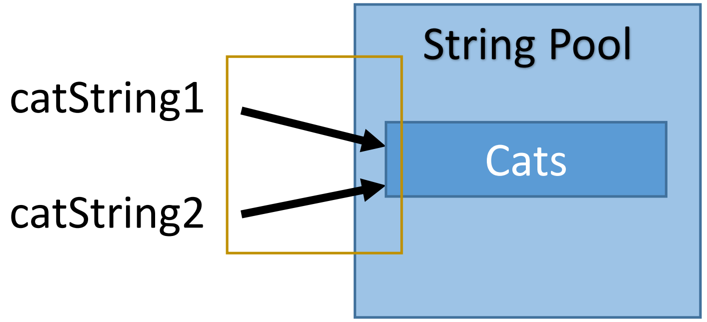
  2. Testing if the *values* of two String variables are the same.



String Comparison (Reference)

- When you compare two String variables (in Java) using the == or != operators, it tests if the same *reference* is the same, not the value.

```
String catString1 = "Cats";
String catString2 = "Cats";

              true
if(catString1 == catString2) {
   System.out.println("The Strings are equal");
}
```



String Pool

catString1

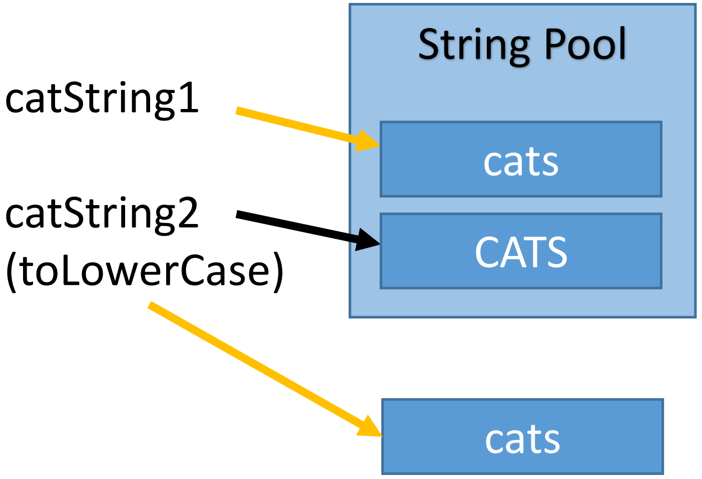
catString2

Cats

String Comparison (Reference)

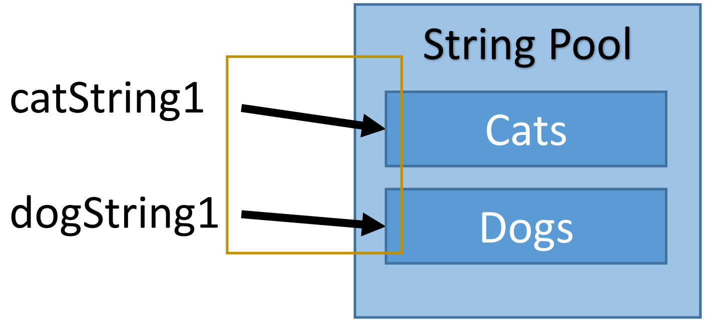
```
String catString1 = "cats";
String catString2 = "CATS";

               false
if(catString1 == catString2.toLowerCase()) {
   System.out.println("The Strings are equal");
}
```

- The references are compared, not the values.



String Comparison (Reference)

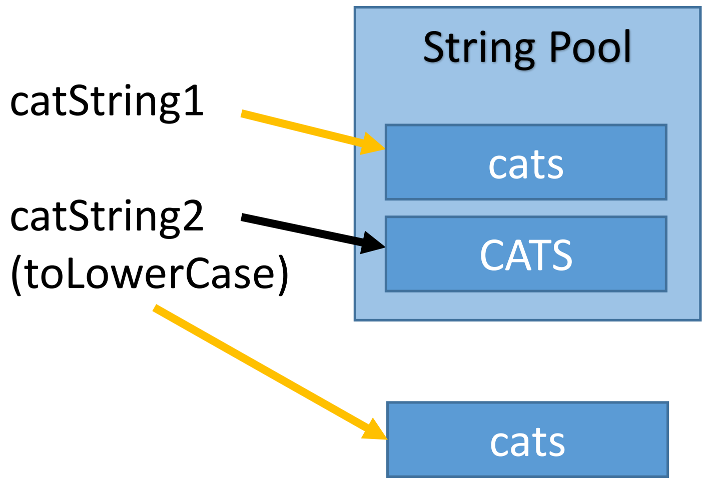
```
String catString1 = "Cats";
String dogString1 = "Dogs";

                   true
if(catString1 != dogString1) {
   System.out.println("The Strings are not equal");
}
```



String Comparison (Reference)

```
String catString1 = "cats";
String catString2 = "CATS";

                  true
if(catString1 != catString2.toLowerCase()) {
   System.out.println("The Strings are not equal");
}
```

- The references are compared, not the values.



String Pool

catString1

cats

catString2

CATS

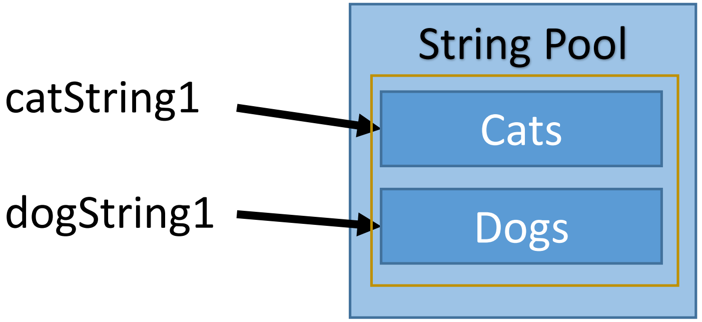
(toLowerCase)

cats

String Comparison (Value)

- To compare the *values* of two Strings, use the String object’s **equals** method.

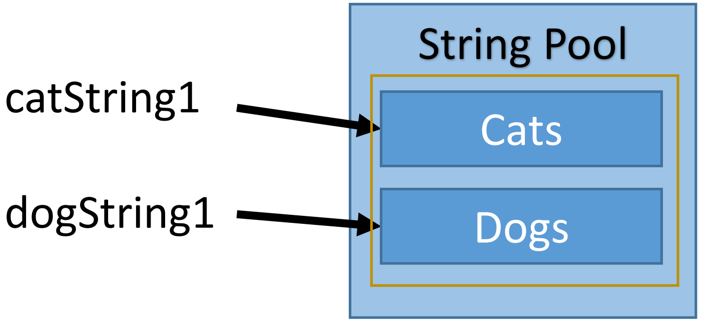
```
String catString1 = "Cats";
String dogString1 = "Dogs";
                false
if(catString1.equals(dogString1)) {
    System.out.println("The Strings are equal");
}
else {
    System.out.println("The Strings are not equal");
}
```



String Comparison (Value)

- To test for inequality, negate the result by using the not operator.

```java
String catString1 = "Cats";
String dogString1 = "Dogs";

if(!catString1.equals(dogString1)) {
    System.out.println("The Strings are not equal");
}
else {
    System.out.println("The Strings are equal");
}
```



String Pool

catString1

Cats

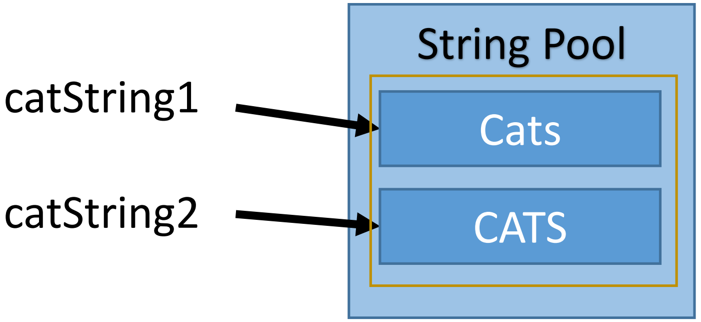
dogString1

Dogs

String Comparison (Value)

- The equals method is case sensitive.
- To ignore upper or lowercase characters when comparing the values of two Strings, use the String object’s **equalsIgnoreCase** method.

```
String catString1 = "Cats";
String catString2 = "CATS";

                  true
if(catString1.equalsIgnoreCase(catString2)) {
    System.out.println("The Strings are equal");
}
else {
    System.out.println("The Strings are not equal");
}
```



String Comparison (Length)

- The length method returns the number of characters in a String.
  - Includes any whitespace.

```
String helloString = "Hello World!";
int totalCharacters = helloString.length();

                   true
if(totalCharacters == 12) {

}
```

String Comparison (Starting text)

- The startsWith method checks to see if the String begins with the value provided.

```
String hello = "Hello World!";

            true
if(hello.startsWith("H")) {

}
```

String Comparison (Starting text)

- The startsWith method is case sensitive.

```
String hello = "Hello World!";

            false
if(hello.startsWith("h")) {

}
```

String Comparison (Starting text)

```
String hello = "Hello World!";

              true
if(hello.startsWith("Hello W")) {

}
```

String Comparison (Ending text)

- Similar to the startsWith method, the endsWith method tests if the String *ends* with a particular character sequence.

```
String hello = "Hello World";

               true
if(hello.endsWith("d")) {

}
```

String Comparison (Ending text)

- The endsWith method is case sensitive.

```
String hello = "Hello World";
```

false

```
if(hello.endsWith("D")) {

}
```

String Comparison (Ending text)

```java
String hello = "Hello World!";
```

false

```java
if(hello.endsWith("World")) {

}
```

String methods

Method	Return Type	Description	Possible Exceptions
equals(String)	boolean	Returns true if the strings are equal, false if not equal; Case sensitive.	None
equalsIgnoreCase(String)	boolean	Returns true if the strings are equal, false if not equal; Case insensitive.	None
length()	int	Returns the length of the String (number of characters; includes symbols and whitespace)	None
startsWith(String)	boolean	Returns true if the String begins with the supplied String, false if it does not.	None
endsWith(String)	boolean	Returns true if the String ends with the supplied String, false if it does not.	None

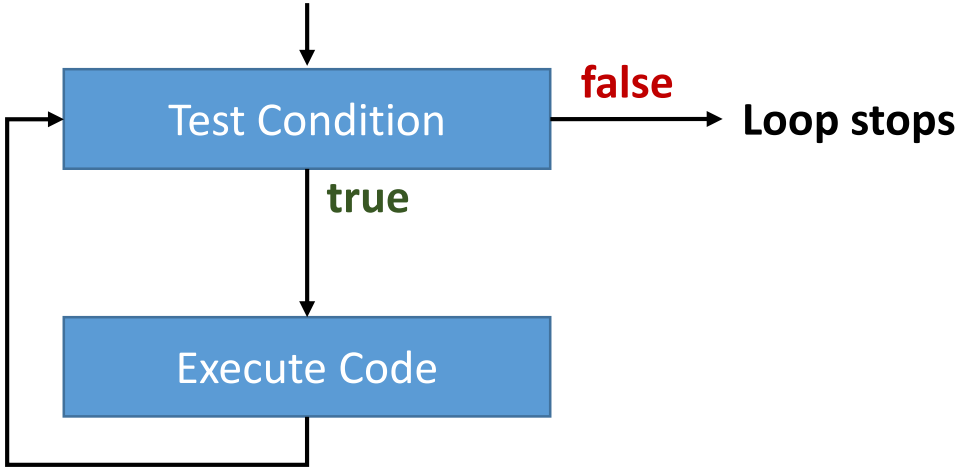
While Loops

- A ***while loop*** repeats as long as its Boolean expression is true.
- The syntax for a Java while loop is shown below.

```
while(Boolean Expression){
  //code that will be
  //executed as long as the
  //Boolean Expression is true
}
```

While Loop (Flow Chart)

- A while loop is a pre-test, sentinel-controlled loop.



While Loop

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number between 1 and 10: ");
int input = Integer.parseInt(keyboard.nextLine());

while(input < 1 || input > 10) {
  System.out.println("Error. Try again.");
  System.out.print("Enter a number between 1 and 10: ");
  input = Integer.parseInt(keyboard.nextLine());
}

System.out.print("Thank you!");
```

```
Enter a number between 1 and 10: 11
Error. Try Again.
Enter a number between 1 and 10: 7
Thank you!
```

While Loop

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter word: ");
String input = keyboard.nextLine();

while(!input.equalsIgnoreCase("exit")) {
  System.out.println("toUpperCase: " + input.toUpperCase());
  //Prompt for input again
  System.out.print("Enter word: ");
  input = keyboard.nextLine();
}

System.out.print("Goodbye!");
```

```
Enter word: cat
toUpperCase: CAT
Enter word: dog
toUpperCase: DOG
Enter word: llama
toUpperCase: LLAMA
Enter word: exit
Goodbye!
```

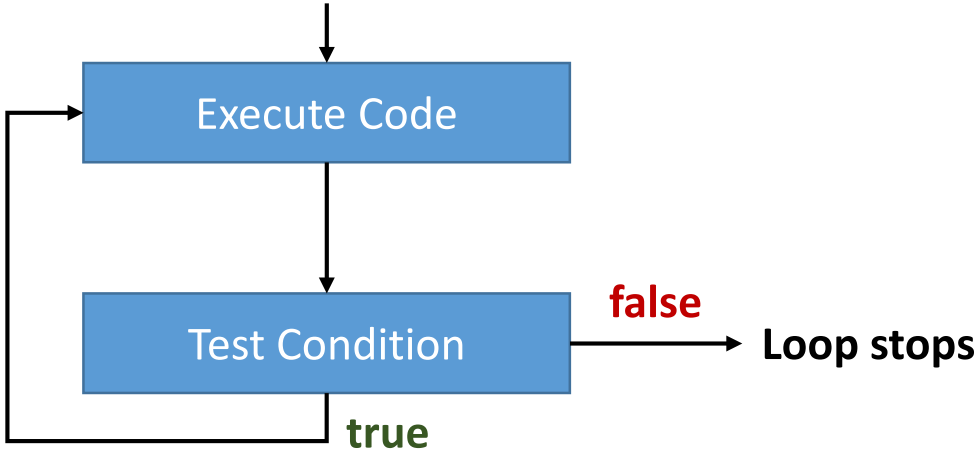
Do-While Loop

- A ***do-while loop*** is a *post-test*, sentinel-controlled loop.
- It will always iterate at least once.
  - Unlike the while loop that tests the condition before the first iteration, the do-while loop tests the condition *after* the first iteration.
- In many cases, the behavior of a do-while loop will be equivalent to the same while loop.

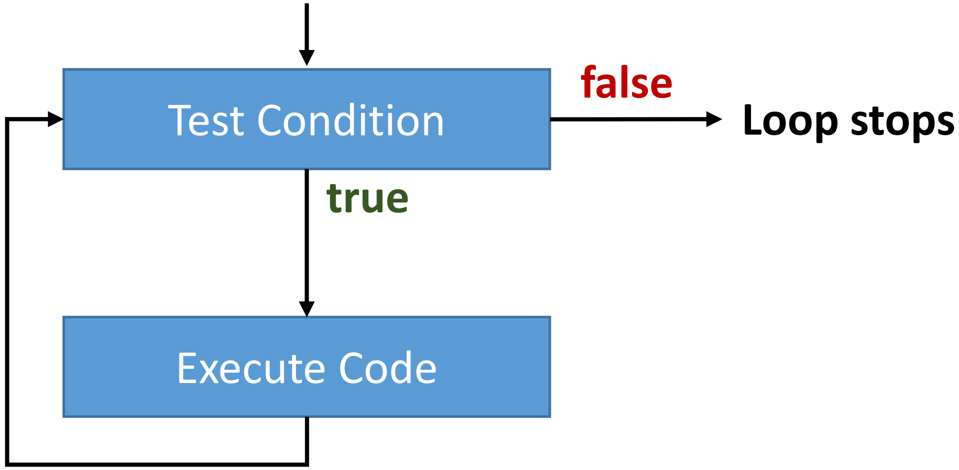
```
do {
   //Code that executes at least once
   //and iterates as long as the
   //condition is true
} while(Boolean expression);
```

Semicolon!

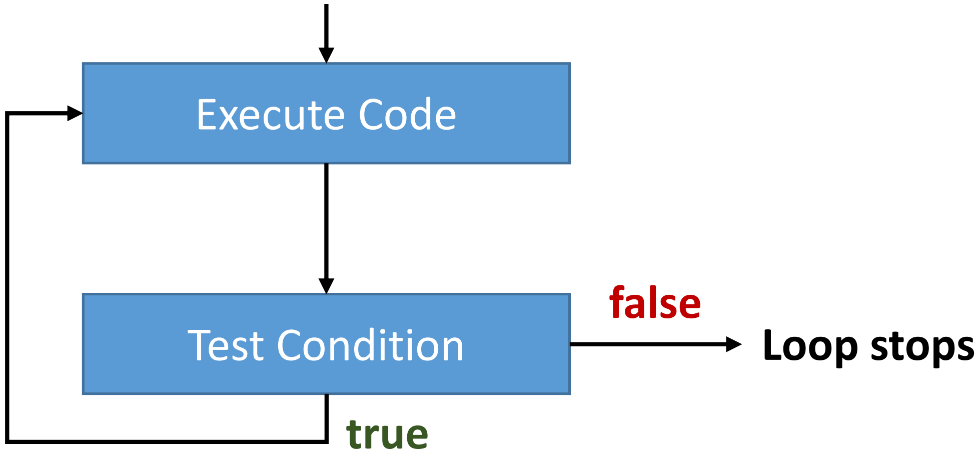
Do-While Loop (Flow Chart)



While vs. Do-While Loop



While Loop



Do-While Loop

Do-While Loop

- This do-while loop verifies that the user’s input was non-negative.

```
Scanner keyboard = new Scanner(System.in);
int sales = 0;
do {
    System.out.print("Enter the total sales for the store: ");
    sales = Integer.parseInt(keyboard.nextLine());
} while(sales < 0);
System.out.print("Thank you.");
```

```
Enter the total sales for the store: -100
Enter the total sales for the store: -5
Enter the total sales for the store: 10
Thank you.
```

Increment (Unary Addition) Operator

- The increment/unary addition operator **++** adds one to the value of a numeric variable.
  - Python does not have this operator.

```
int testNumber = 5;
testNumber++; //Value of testNumber is now 6
```

Increment (Unary Addition) Operator

- The increment operator can come before the variable name (prefix) or after the variable name (postfix).
- Both increment the variable by one.

```
int testNumber = 5;
```

- Prefix:

```
++testNumber;
```

- Postfix:

```
testNumber++;
```

Prefix Unary Addition

- With prefix, 1 will be added **before** the value is returned.
  - This usually will only matter when you are performing the increment as you assign the value to another variable.
  - Example:

```
int testNumber = 5;
int otherNumber = ++testNumber;
```

- In the second line…
  - 1 will be added to testNumber, making the value of testNumber to be 6
  - This new value of 6 will be assigned to otherNumber.

Postfix Unary Addition

- With postfix, 1 will be added **after** the value is returned.
  - Example:

```
int testNumber = 5;
int otherNumber = testNumber++;
```

- In the second line…
  - The value of testNumber, which is 5, is assigned to otherNumber.
  - 1 is then added to testNumber, making the value of testNumber 6.

Decrement (Unary Subtraction) Operator

- The decrement/unary subtraction operator **--** subtracts one from the value of a numeric variable.
  - Python doesn't have this operator, either.

```
int testNumber = 5;
testNumber--; //Value of testNumber is now 4
```

Decrement (Unary Subtraction) Operator

- The decrement operator can come before the variable name (prefix) or after the variable name (postfix).
- Both decrement the variable by one.

```
int testNumber = 5;
```

- Prefix:

```
--testNumber;
```

- Postfix:

```
testNumber--;
```

Prefix Unary Subtraction

- With prefix, 1 will be subtracted **before** the value is returned.
  - This usually will only matter when you are performing the decrement as you assign the value to another variable.
  - Example:

```
int testNumber = 5;
int otherNumber = --testNumber;
```

- In the second line…
  - 1 will be subtracted from testNumber, making the value of testNumber 4
  - This new value of 4 will be assigned to otherNumber.

Postfix Unary Subtraction

- With postfix, 1 will be subtracted **after** the value is returned.
  - Example:

```
int testNumber = 5;
int otherNumber = testNumber--;
```

- In the second line…
  - The value of testNumber, which is 5, is assigned to otherNumber.
  - 1 is then subtracted from testNumber, making the value of testNumber 4.

Increment and Decrement Operators

- To recap:
    - Prefix increment/decrement: 1 is added/subtracted **before** the value is returned or used.
    - Postfix increment/decrement: 1 is added/subtracted **after** the value is returned or used.

- If you just want to add or subtract 1 to/from a numeric value, pre/postfix doesn't matter.

For Loops

- A ***for loop*** is a pre-test, count-controlled loop.

- Java has two types of for loops:
  - An enhanced for loop (Like Python’s)
  - A traditional (“C-Style”) for loop.

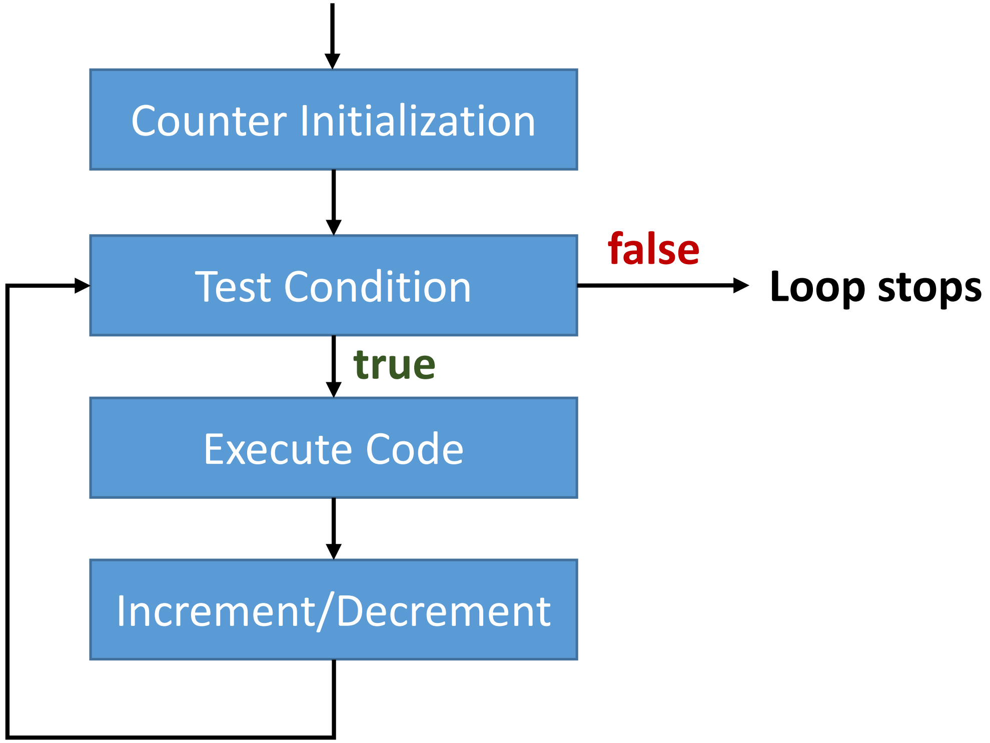
- Java’s enhanced for loop will be demonstrated in a future lecture.

For Loop

- A traditional for loop has three parts, separated by semicolons:
  - Initialization- Declares an int variable to be used as a *control counter*.
  - Termination Condition- A Boolean expression tested at the beginning of each iteration.
    - If true, the loop's code executes; If false, the loop stops.
  - Increment/Decrement- Happens at the end of each iteration; Normally increments or decrements the control counter.

```
for(initialization; termination; increment/decrement) {
   //Code that executes each iteration
}
```

For Loop (Flow Chart)



For Loop

Initialization- Here, we have initialized an int (named “counter”) to the value 1.

Termination- As long as counter is less than or equal to 5, the loop will iterate again.

Increment- At the end of the iteration, add 1 to the value of counter.

```
for(int counter = 1; counter <= 5; counter++) {
   System.out.println("Lap #" + counter);
}
System.out.println("Finished!");
```

Note- The “counter” variable is only accessible *inside* the loop.

For Loop

```java
for(int counter = 1; counter <= 5; counter++) {
   System.out.println("Lap #" + counter);
}
System.out.println("Finished!");
```

```
Lap #1
Lap #2
Lap #3
Lap #4
Lap #5
Finished!
```

For Loop

```
for(int i = 3; i <= 7; i++) {
   System.out.println("Number:  " + i);
}
```

```
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
```

For Loop

```java
for(int i = 3; i >= 0; i--) {
   System.out.println("Number: " + i);
}
```

```
Number: 3
Number: 2
Number: 1
Number: 0
```

For Loop

Unlike previous examples that increment or decrement by one, this example shows that we can increment or decrement by a larger step.

```java
for(int i = 2; i < 10; i += 2) {
   System.out.println("Number: " + i);
}
```

```
Number: 2
Number: 4
Number: 6
Number: 8
```

For, While, and Do-While Loops

- “C-Style”/Traditional For Loops
  - Pre-test, count-controlled.
  - Use when you need to iterate over a range of numbers.

- While Loop
  - Pre-test, sentinel-controlled.
  - Use when you need to iterate as long as a condition is and remains true.

- Do-While Loop
  - Post-test, sentinel-controlled.
  - Use when you need to iterate at least one time and possibly more times.

Branching Statements

- There are two branching statements that allow us to either:
  - Immediately exit a loop.
  - Immediately begin the next iteration.

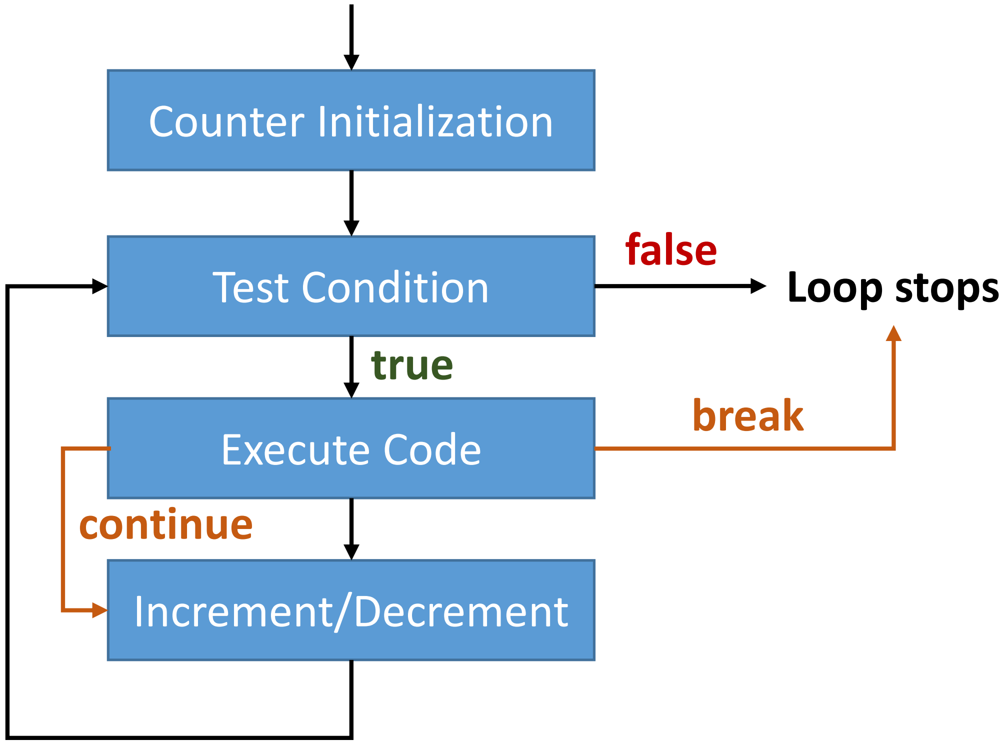
```
break;
```

- We have already seen the break statement when using a switch.
- It works in a similar fashion in a loop. Once encountered, the loop will immediately stop where it is. The code following the loop structure will begin to be executed.

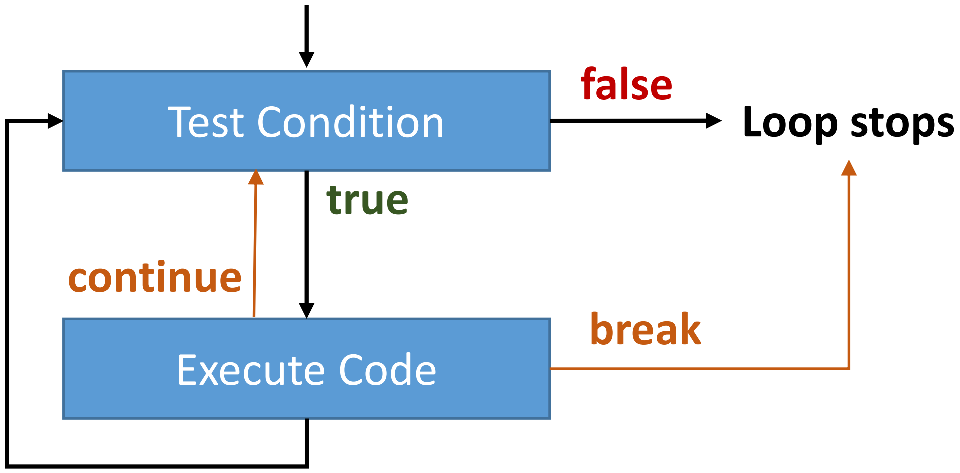
```
continue;
```

- Once encountered, the loop will immediately stop where it is and begin the next iteration.

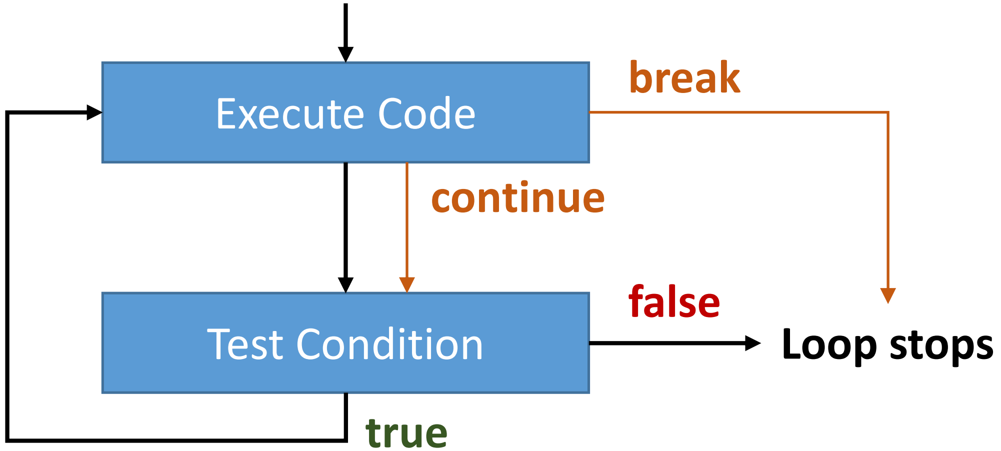
For Loop (Updated Flow Chart)



While & Do-While Loop (Flow Charts)



While Loop



Do-While Loop

break statement

```
for(int myInt = 1; myInt < 11; myInt++) {
  if(myInt > 5) {
    break;
  }
  System.out.println("Number: " + myInt);
}
System.out.println("All done!");
```

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
All done!
```

- This loop normally would have printed “Number: 1” through “Number: 10”
- However, once the value of myInt is greater than 5, the break statement will be encountered.
- The loop will exit immediately and resume the code outside of the loop.

continue statement

```java
for(int myInt = 2; myInt <= 11; myInt++) {
  if(myInt % 2 == 1) {
    continue;
  }
  System.out.println("Number: " + myInt);
}
System.out.println("All done!");
```

```
Number: 2
Number: 4
Number: 6
Number: 8
Number: 10
All done!
```

- If myInt is odd, the continue statement will be encountered.
- Instead of finishing the iteration, the loop begins the next iteration.

Nested Loops

- A nested loop is a loop within a loop.
- For every iteration of the outer loop, the inner loop will be iterated to completion.

```
for(int row = 1; row <= 5; row++) {

  for(int column = 1; column <= row; column++) {
    System.out.print("#");
  }
  System.out.println();

}
```

```
#
##
###
####
#####
```

Be sure to use different names for your counters. Any variables declared in outer loops will be accessible by inner loops, including the outer loop’s counter.

Infinite For Loops

- An infinite loop is a loop that does not stop or exit.
- In many cases, an infinite loop is the result of poor programming.

```
for(int i = 1; i <= 10; i++) {
   i--;
   System.out.println("Number: " + i);
}
```

```
Number: 0
Number: 0
Number: 0
…
```

```
for(int i = 1; i <= 10; i--) {
   System.out.println("Number: " + i);
}
```

```
Number: 1
Number: 0
Number: -1
Number: -2
…
```

Infinite While Loops

```java
boolean done = false;
int myInt = 0;
while(!done) {
  myInt++;
  System.out.println("Number: " + myInt);
}
```

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
…
```

Infinite Loops

- Sometimes, infinite loops can be useful.
  - For example, perpetually getting user input until they enter a command to exit or a valid entry.

- However, when we intentionally create an infinite loop, we will want to provide some way for the loop to exit.
  - Use a break statement to stop the loop.

“For-ever” Statement

- A for loop with no initialization, termination, or increment creates an infinite loop colloquially called a “for-ever loop”.

```
System.out.println("Forever");
for(;;) {
  System.out.println("and ever");
}
```

```
Forever
and ever
and ever
and ever
and ever
…
```

“For-ever” Loop

```
for(;;) {
  System.out.print("Enter a command: ");
  String command = keyboard.nextLine();
  if(command.equalsIgnoreCase("Exit")) {
    break;
  }
  else {
    System.out.println("You entered: " + command);
  }
}
```

Infinite While Loop

```
while(true) {
  System.out.print("Enter a number (0 to Exit): ");
  numberToSquare = Integer.parseInt(keyboard.nextLine());
  if(numberToSquare == 0) {
    break;
  }
  else {
    System.out.println("Your number squared is: " +
                       Math.pow(numberToSquare, 2));
  }
}
```

Infinite Do-While Loop

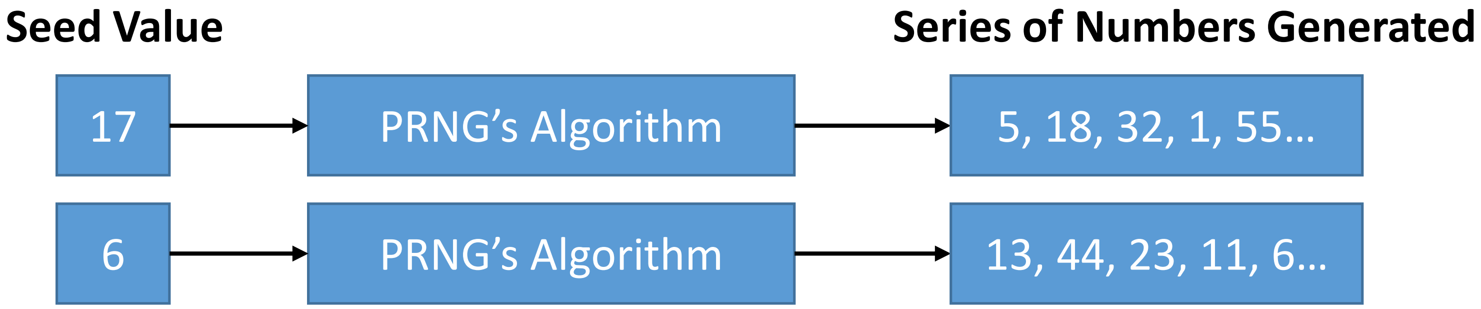
```
do {
  System.out.print("Enter a number (0 to Exit): ");
  numberToSquare = Integer.parseInt(keyboard.nextLine());
  if(numberToSquare == 0) {
    break;
  }
  else {
    System.out.println("Your number squared is: " +
                                   Math.pow(numberToSquare, 2));
  }
} while(true);
```

Random Number Generators

- A ***random number generator*** is software or hardware that produces a random number.
    - A ***random number*** is number chosen from a set of possible values, each with the same probability of being selected.
- A Pseudo-Random Number Generator (PRNG) uses a mathematical algorithm to generate a series of seemingly random numbers.
    - Software Generators
- A True Random Number Generator (TRNG) uses an unpredictable physical means to generate random numbers.
    - Hardware Generators

Random Number Generators

- As stated, PRNGs use an algorithm to generate the series of numbers.
- A ***seed*** is a number provided to a PRNG as an input to its algorithm.



- Using the same seed will produce the same series of numbers.
  - If you know how the PRNG's algorithm works and the seed that's being used, you will know the series of numbers it will generate.
  - Hence why it is pseudo-random.

Random Object

- Java’s Random object is used as a PRNG.

```
import java.util.Random;

public class RandomNumberGenerator {

    public static void main(String[] args) {
        //Create a new instance of the Random object.
        //Uses a seed generated by the JVM.
        Random myGenerator = new Random();

        //Assigns a random number between 0 and 4 to someNumber.
        int someNumber = myGenerator.nextInt(5);
    }

}
```

Import the Random object from java.util
The Random object can be used as a Random Number Generator

Random Object

- Must be imported.

```
import java.util.Random;
```

- Must be instantiated.

```
Random myGenerator = new Random();
```

Random Object

- The nextInt() method accepts one int argument
  - Returns a number from the range from zero up to, but not including, the argument's value.
- Draws a random number between 0 and 9:

```
int someNumber = myGenerator.nextInt(10);
```

- Draws a random number between 0 and 100:

```
int someNumber = myGenerator.nextInt(101);
```

Random Object

- Draws a random number between 1 and 5:

```
int someNumber = myGenerator.nextInt(5)+1;
```

Total Numbers

Start Value

- Results:

```
myGenerator.nextInt(5)   → 0, 1, 2, 3, or 4
myGenerator.nextInt(5)+1 → 1, 2, 3, 4, or 5
```

Range of possible numbers
that could be generated

Random Object

- Draws a random number between 21 and 29:

```
int someNumber = myGenerator.nextInt(9)+21;
```

Total Numbers

Start Value

- Results:

```
myGenerator.nextInt(9)    ──────→   0,  1,  2,  3,  4,  5,  6,  7,  8

myGenerator.nextInt(9)+21 ──────→  21, 22, 23, 24, 25, 26, 27, 28, 29
```

Range of possible numbers
that could be generated

Random Object

- An argument can be provided at instantiation.
  - Will act as the generator's seed value.

- However, this will always generate the same series of numbers every time.
  - The generator's algorithm doesn't change.
  - If the seed remains the same, the algorithm will produce the same output.

Random Object

```java
import java.util.Random;

public class RandomNumberGenerator {

    public static void main(String[] args) {
        //Create a new instance of the Random object.
        //Uses a supplied seed.
        Random myGenerator = new Random(1034);

        //Assigns a random number between 0 and 4 to someNumber.
        int someNumber = myGenerator.nextInt(5);
    }

}
```