

# Repetitive Structures

Michael C. Hackett

Associate Professor, Computer Science



# Lecture Topics

- Repetitive Structures
  - Count-Controlled Loops (For)
  - Unary Addition/Subtraction
  - Sentinel-Controlled Loops (While/Do-While)
- Nested 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

Boolean expression is false

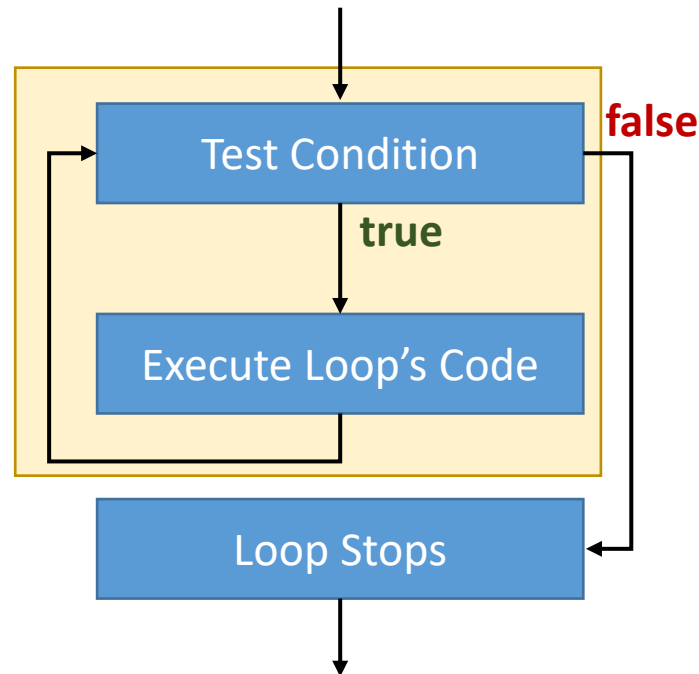
Boolean expression is true

# Loops

- A ***loop*** is a programming structure that allows code to be repeatedly executed, usually as long as some condition (Boolean expression) evaluates to true.
  - Each repetition of the loop's code is called an ***iteration***.
- Programming languages have a few types of loops.
  - Pre-test and Post-test Loops
  - Sentinel-Controlled and Count-Controlled.

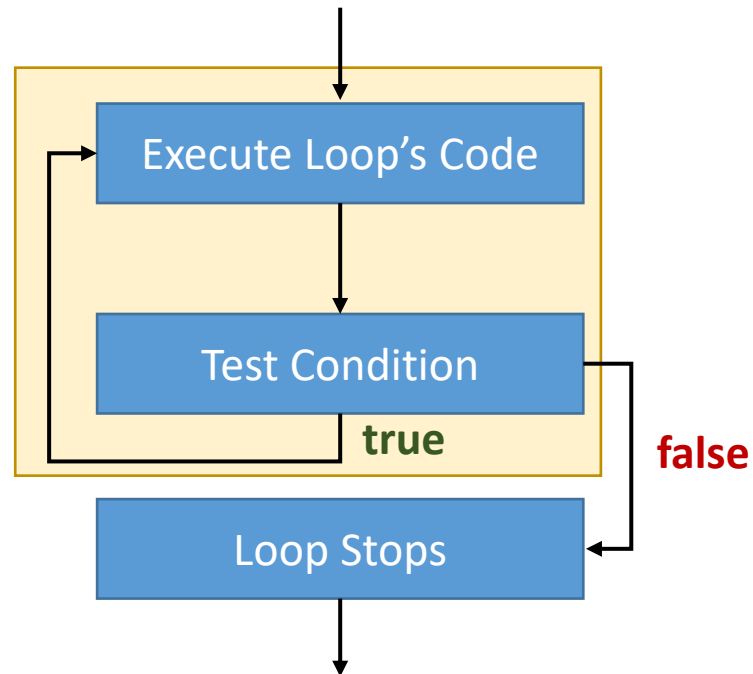
# Types of Loops

- ***Pre-test loops*** test the condition *before* starting each iteration.



# Types of Loops

- ***Post-test loops*** test the condition *after* completing each iteration.



# For Loops

- A ***for loop*** is a pre-test, count-controlled loop.
- Java has two types of for loops:
  - A traditional (“C-Style”) for loop.
  - An “enhanced” for loop.
- The enhanced for loop will be demonstrated in a future lecture.

# For Loops

- 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 Loops

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

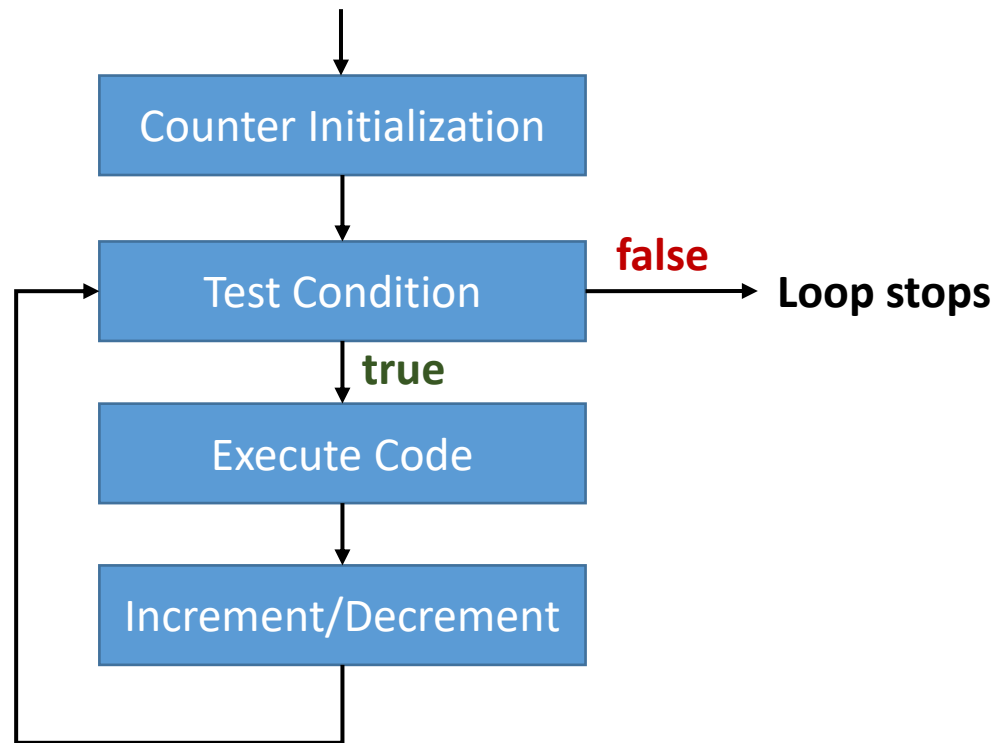
Termination- As long as counter is less than or equal to 5, the loop will iterate again.

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 Loops



# Increment (Unary Addition) Operator

- The increment/unary addition operator `++` adds one to the value of a numeric variable.

```
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.

```
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

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

Condition- 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("Iteration #" + counter);  
}  
System.out.println("Finished!");
```

No semicolon!

Note- The value of "counter" is only accessible *inside* the loop.

# For Loops

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

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

# For Loops – Use caution when not using braces

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



No semicolon!

- The example above works identically as the previous example, even though there are no curly braces.
- When there are no curly braces, the for loop operates on only the next line.
- It is good to always use braces though; it makes it easier to read and see exactly what code is included in the loop and what code is not.

# For Loops

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

What is the output?

# For Loops – Decrement Example

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

Number: 3

Number: 2

Number: 1

Number: 0



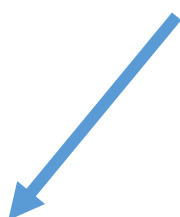
# For Loops

```
for(int i = 10; i >= 3; i--) {  
    System.out.println("Number: " + i);  
}
```

What is the output?

# For Loops

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



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

Number: 2

Number: 4

Number: 6

Number: 8

# For Loops

```
for(int i = 3; i <= 2; i--) {  
    System.out.println("Number: " + i);  
}
```

- What is the output?

# For Loops

```
for(int i = 3; i <= 2; i--) {  
    System.out.println("Number: " + i);  
}
```

- There is no output!
- Remember, the Boolean condition is tested at the beginning of each iteration.
- 3 is NOT less than or equal to 2, so the condition is false; The loop doesn't even iterate once.

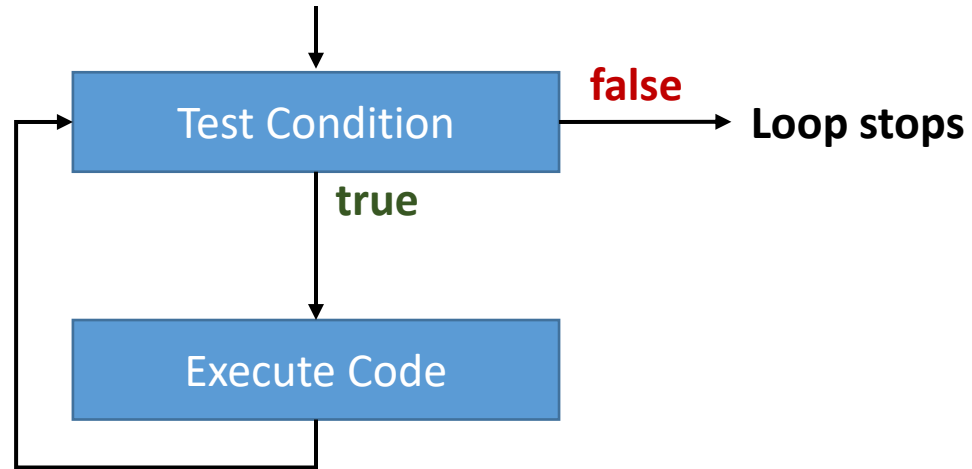
# While Loops

- A ***while loop*** repeats as long as its Boolean expression is true

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

# While Loops

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



# While Loops

```
Scanner keyboard = new Scanner(System.in);
String input = ""; //Will hold the user's entry
System.out.print("Enter word: ");
input = keyboard.nextLine();
while(!input.equals("exit")) {
    //Print the input in uppercase
    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!
```

# While Loops

- This next example will allow the user to enter a number and the program will display the number, squared.
- When the user enters 0, the program will stop.



# While Loops

```
Scanner keyboard = new Scanner(System.in);
boolean done = false;

int numberToSquare = 0;

while(!done) {
    System.out.print("Enter a number: ");
    numberToSquare = Integer.parseInt(keyboard.nextLine());
    if(numberToSquare < 1) {
        done = true;
    }
    else {
        System.out.println("Your number squared is: " + Math.pow(numberToSquare, 2));
    }
}
```

# While Loops

```
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!
```

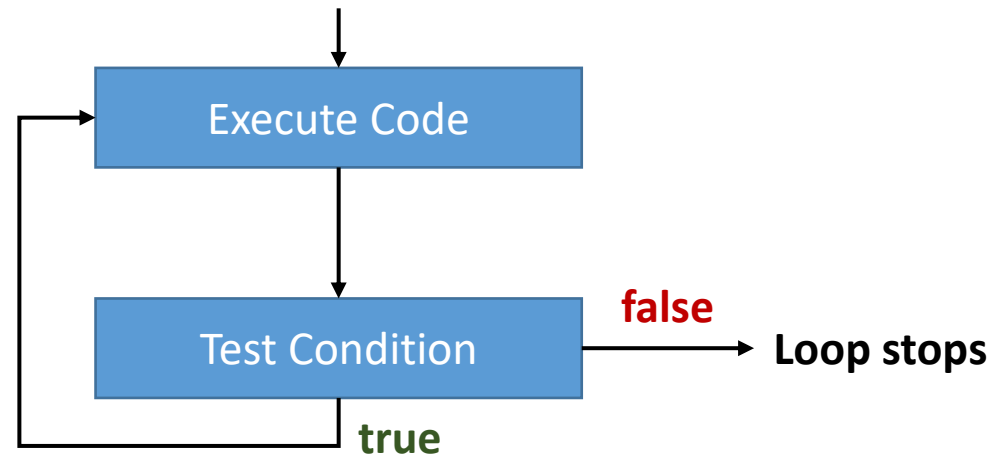
# Do-While Loops

- 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.

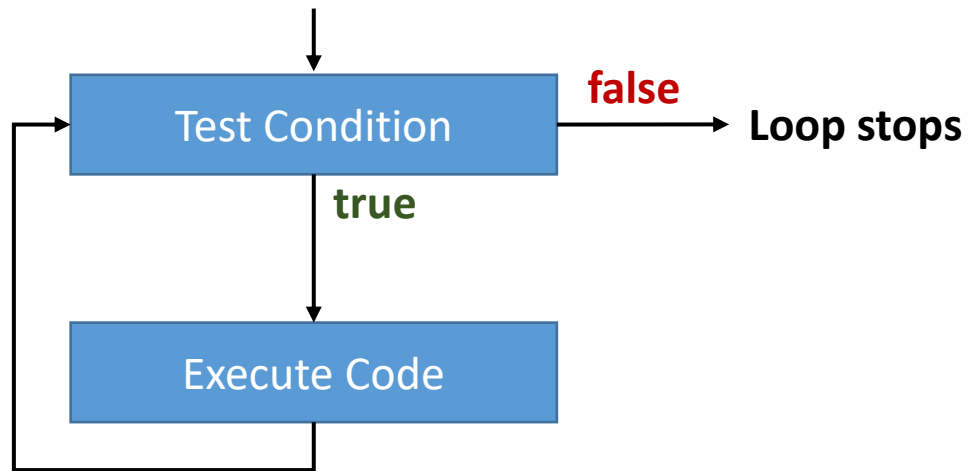
```
do {  
    //Code that executes at least once  
    //and iterates as long as the  
    //condition is true  
} while(Boolean expression);
```

← Semicolon!

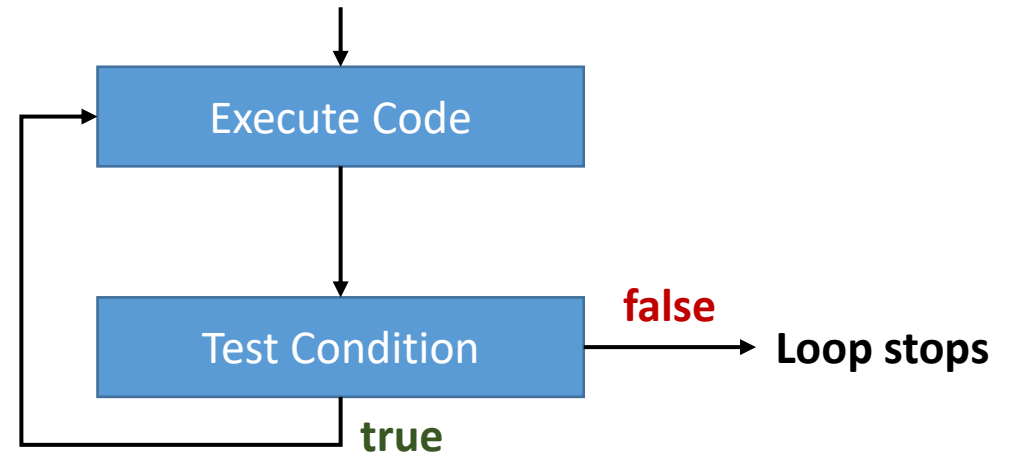
# Do-While Loops



# Do-While Loops



While Loop



Do-While Loop

# Do-While Loops

- 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.
```

# When to use a While or Do-While Loop

- In many cases, it won't matter which you use.
- Use a Do-While loop when you want to *guarantee* the loop to iterate at least once, before the boolean condition is ever tested.

# 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 i = 1; i <= 5; i++) {  
    System.out.println("Number: " + i);  
    for(int j = 1; j <= 3; j++) {  
        System.out.println("Number: " + j);  
    }  
}
```

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



# Nested Loops

```
for(int i = 1; i <= 5; i++) {  
    System.out.println("Number: " + i);  
    for(int j = 1; j <= 3; j++) {  
        System.out.println("Number: " + j);  
    }  
}
```

Number: 1		
Number: 1	Inner Loop	Outer Iteration 1
Number: 2		
Number: 3		
Number: 2		
Number: 1	Inner Loop	Outer Iteration 2
Number: 2		
Number: 3		
Number: 3		
Number: 1	Inner Loop	Outer Iteration 3
Number: 2		
Number: 3		
Number: 4		
Number: 1	Inner Loop	Outer Iteration 4
Number: 2		
Number: 3		
Number: 5		
Number: 1	Inner Loop	Outer Iteration 5
Number: 2		
Number: 3		

# Nested Loops – Variable scope

```
for(int i = 1; i <= 3; i++) {  
    String str = "TestString";  
    for(int j = 1; j <= 3; j++) {  
        System.out.println(str + i);  
    }  
}
```

```
TestString1  
TestString1  
TestString1  
TestString2  
TestString2  
TestString2  
TestString3  
TestString3  
TestString3
```

- Even though the String “str” and the counter “i” are declared and assigned outside of the inner loop, the inner loop is still able to access them.
- This does not work both ways! Variables declared in inner loops are not accessible by outer loops.

# 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 for a loop. Once encountered, the loop will immediately stop where it is. Any code outside/after of the loop will begin to be executed.

## **continue;**

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

# break statement – for loop

```
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.
- Works the same way in a while/do-while loop.

# continue statement – for loop

```
for(int myInt = 1; 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 and printing out the number, the loop stops there and restarts.
- In the case of a for loop, the continue statement will increment/decrement the counter as in a normal iteration.
- In the case of a while/do-while loop, the continue statement will force the next iteration; the condition will still be tested before the next iteration begins, as normal.

# Infinite Loops

- An infinite loop is a loop that does not stop or exit.
- The only way to stop an infinite loop (that is the result of poor design) is to forcibly stop the program.
- In most cases, an infinite loop is the result of poor programming.
- However, sometimes programmers intentionally create infinite loops.

# Infinite Loops

- The two biggest culprits for unintentional, program-crashing infinite loops are:
  - Forgetting to change a value that the Boolean condition of a while/do-while loop depends on.
    - Hence, the condition stays true and never evaluates as false.
  - Changing the increment/decrement counter's value within the body of its for loop.

# Infinite While Loop

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

Number: 1

Number: 2

Number: 3

Number: 4

Number: 5

...

- Since the value of “done” is never changed in the body of the loop, there is no way the Boolean condition will ever be false.



# Infinite For Loop

```
for(int i = 1; i <= 15; i++) {  
    i--;  
    System.out.println("Number: " + i);  
}
```

Number: 0

Number: 0

Number: 0

Number: 0

Number: 0

...

- Even though “i” is incremented by 1 after every iteration, it is first being decremented by 1 in the body of the loop.
- There is no way the Boolean condition will ever be false, so there is no way this loop would ever end.

# Infinite Loops

- With some care and sound logic, infinite loops can be useful.
  - For example, perpetually getting user input until they enter a command to exit (similar to earlier examples)
- However, when we intentionally create an infinite loop, we normally will want to provide some way for the loop to exit.
  - Use a break statement to stop the loop.

# For-ever Loop

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

```
Forever  
and ever  
and ever  
and ever  
and ever  
...
```

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

# For-ever Loop – Exponents Program

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

# Infinite While Loop

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

# Random Number Generators

- A ***random number*** is number chosen from a set of possible values, each with the same probability of being selected.
- A ***random number generator*** is software or hardware that produces a random number.
- A ***seed*** is a number provided to an algorithm to produce strings of random numbers.

# Types of Random Number Generators

- A Pseudo-Random Number Generator (PRNG) uses a mathematical algorithm to generate random numbers.
- A True Random Number Generator (TRNG) uses an unpredictable physical means to generate random numbers.
  - Atmospheric noise (Using radio waves of atmospheric disturbances)
  - Nuclear decay radiation (Using Geiger counters)
  - Lava lamps (.....Wait, what?)

# Example

Import the Random object from java.util

The Random object can be used as a Random Number Generator

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

```
public class RandomNumberGenerator {
```

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

```
        //Create a new instance of the Random object.
```

```
        //Uses a random seed generated by the JVM.
```

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

```
        //Assigns a random number between 0 and 4 to someNumber.
```

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

```
    }
```

```
}
```



# Example – User Supplied Seed

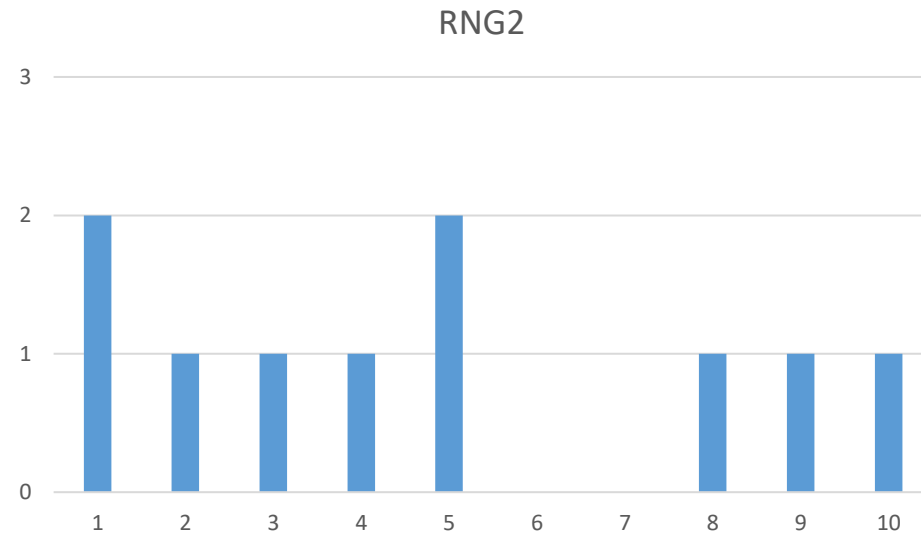
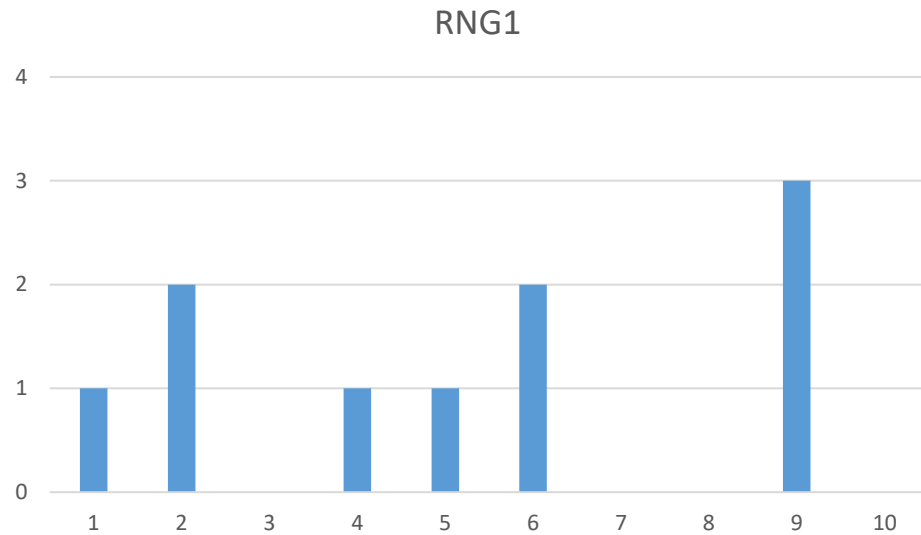
```
import java.util.Random;

public class RandomNumberGenerator {

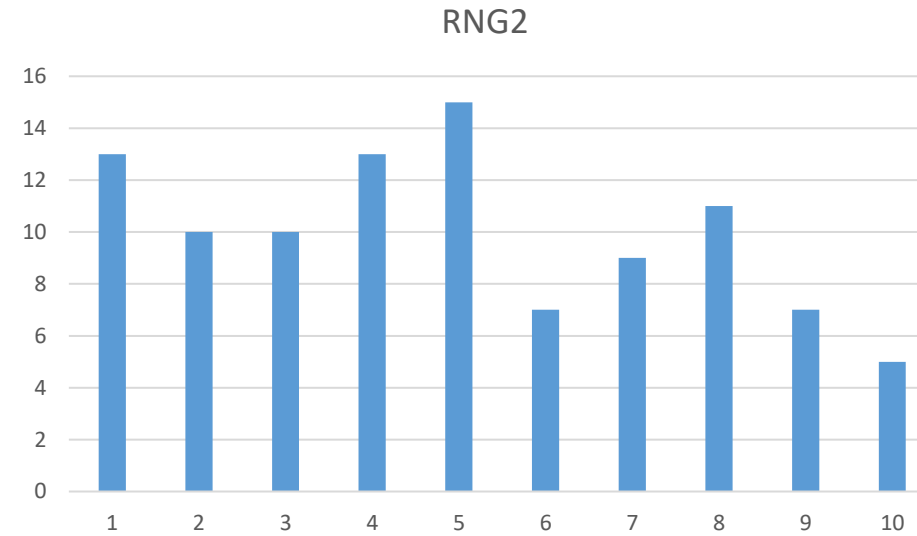
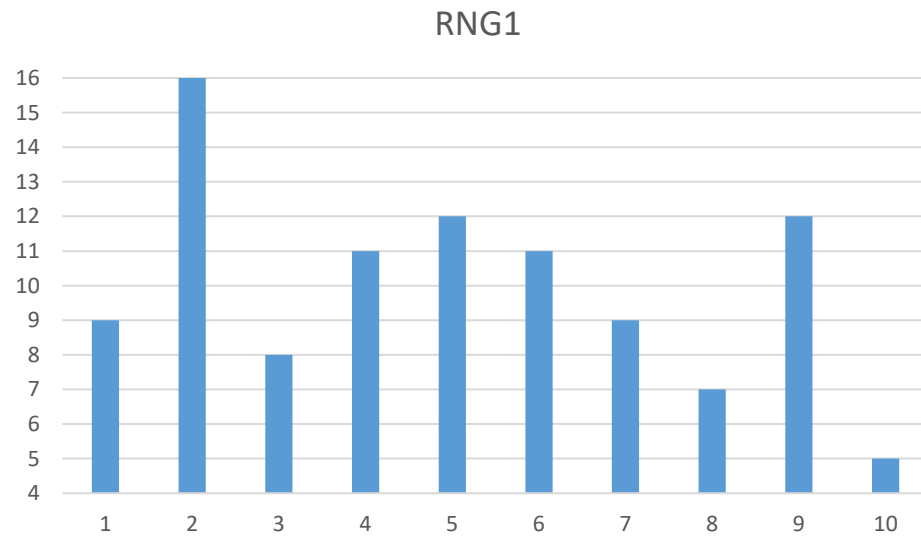
    public static void main(String[] args) {
        //Create a new instance of the Random object.
        //Uses a supplied seed.
        Random myGenerator = new Random(1034);

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

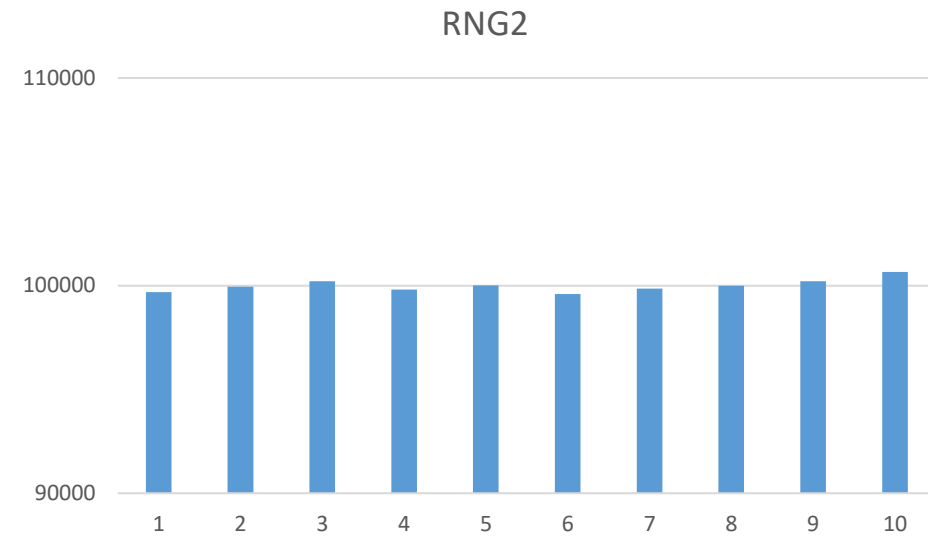
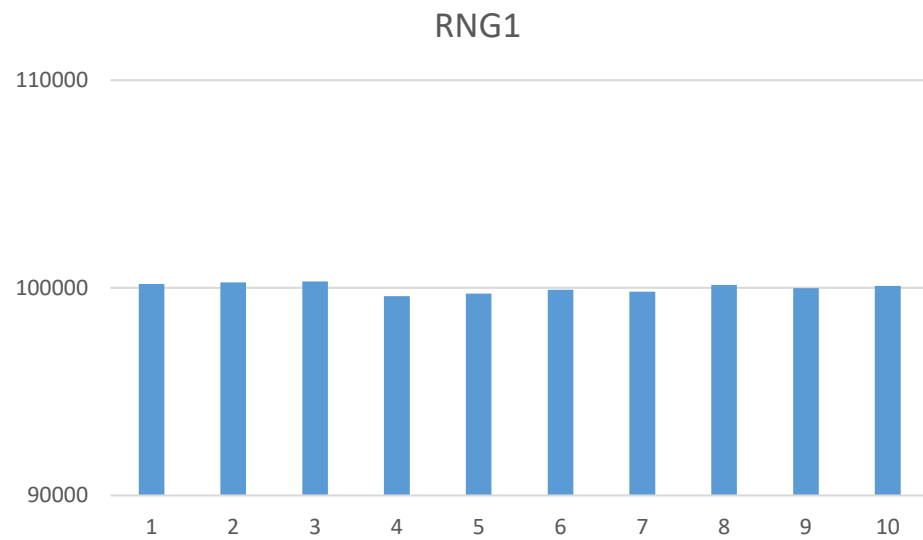
# Evaluating Random Number Generators



# Evaluating Random Number Generators



# Evaluating Random Number Generators



# Rolling dice

- Have a Random Number Generator pick two numbers between one and six.
- Sum the two numbers.

# Example

```
import java.util.Random;

public class DiceRoller {

    public static void main(String[] args) {

        //Instantiate a new Random object.
        Random myGenerator = new Random();

        //Assigns a random number between 1 and 6 to firstDieNumber.
        int firstDieNumber = myGenerator.nextInt(6) + 1;

        //Assigns a random number between 1 and 6 to secondDieNumber.
        int secondDieNumber = myGenerator.nextInt(6) + 1;

        //Calculates the sum of the two "rolls".
        int rollTotal = firstDieNumber + secondDieNumber;

    }
}
```

# Rolling dice

