Revised: 1/8/2021

Object-Oriented Programming IV

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

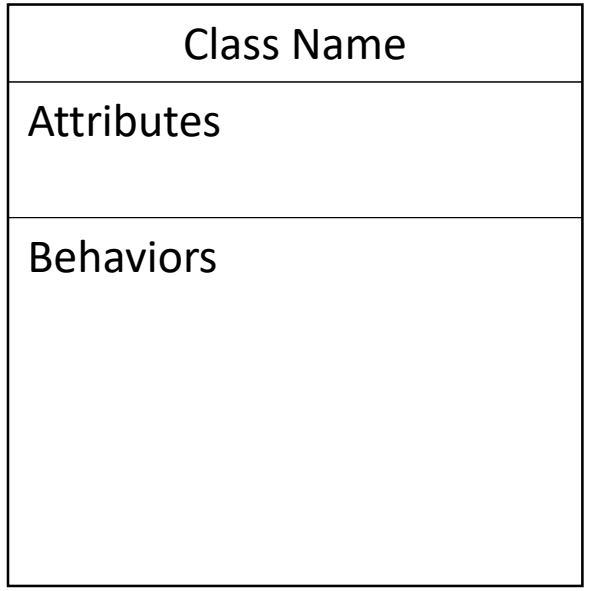
Lecture Topics

- Class Diagrams
- Repetitive Structures
  - While Loops
  - Do While Loops
  - Unary Increment/Decrement
  - For Loops
- Aggregation
- Random Number Generators
- Reading and Writing Text Files

- Extras
  - Nested Loops
  - Infinite Loops

Class Diagrams

- Unified Modeling Language (UML) provides a set of standard diagrams for graphically depicting an object-oriented system.
- In UML, each class is shown as a box, with three sections:
  - The Class Name
  - Class Attributes (Fields)
  - Class Behaviors (Constructors and Methods)



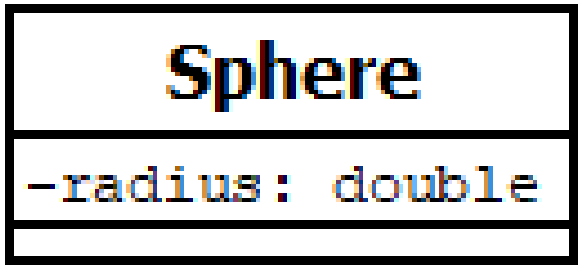
Class Diagrams

- When displaying fields (and parameter names in methods) in a class diagram, the format to use is:
  - name : type

- Access specifier symbols:
  - \+ public fields/methods
  - \- private fields/methods
  - None (or ~) No access specifier.

Class Diagram (Fields)

```
public class Sphere {

      private double radius;

      public Sphere(double r) {
            validateRadius(r);
      }

      public void setRadius(double r) {
            validateRadius(r);
      }

      public void setRadius(String r) {
            validateRadius(Double.parseDouble(r));
      }

      private void validateRadius(double r) {
            if(r > 0) {
                  radius = r;
            }
            else {
                  radius = 1;
            }
      }

      public double getRadius() {
            return radius;
      }
}
```



Class Diagrams (Constructors and Methods)

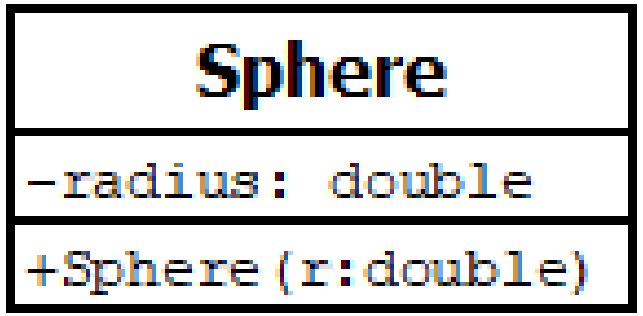
- When displaying a constructor in a class diagram, the format to use is:
  - name(arg : type, …)

- When displaying a method in a class diagram, the format to use is:
  - name(arg : type, …) : returnType

- Access modifier symbols used are the same as for fields.

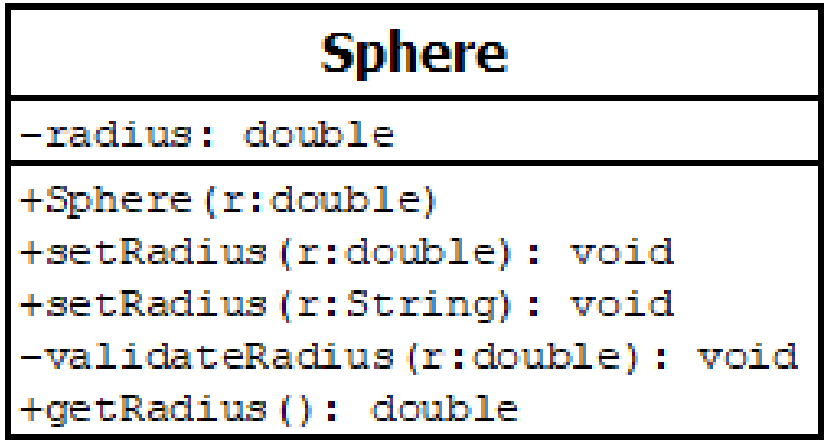
Class Diagram (Constructors)

```
public class Sphere {

	private double radius;

	public Sphere(double r) {
		validateRadius(r);
	}

	public void setRadius(double r) {
		validateRadius(r);
	}

	public void setRadius(String r) {
		validateRadius(Double.parseDouble(r));
	}

	private void validateRadius(double r) {
		if(r > 0) {
			radius = r;
		}
		else {
			radius = 1;
		}
	}

	public double getRadius() {
		return radius;
	}
}
```



Class Diagram (Methods)

```
public class Sphere {

	private double radius;

	public Sphere(double r) {
		validateRadius(r);
	}

	public void setRadius(double r) {
		validateRadius(r);
	}

	public void setRadius(String r) {
		validateRadius(Double.parseDouble(r));
	}

	private void validateRadius(double r) {
		if(r > 0) {
			radius = r;
		}
		else {
			radius = 1;
		}
	}

	public double getRadius() {
		return radius;
	}
}
```



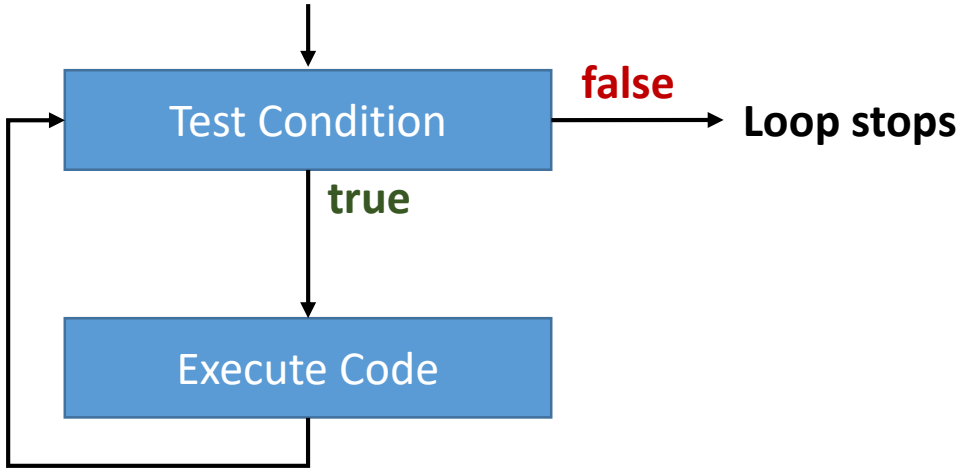
Repetitive Structures

- 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
}
```

Repetitive Structures

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



Repetitive Structures

```
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!
```

Repetitive Structures

```
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!
```

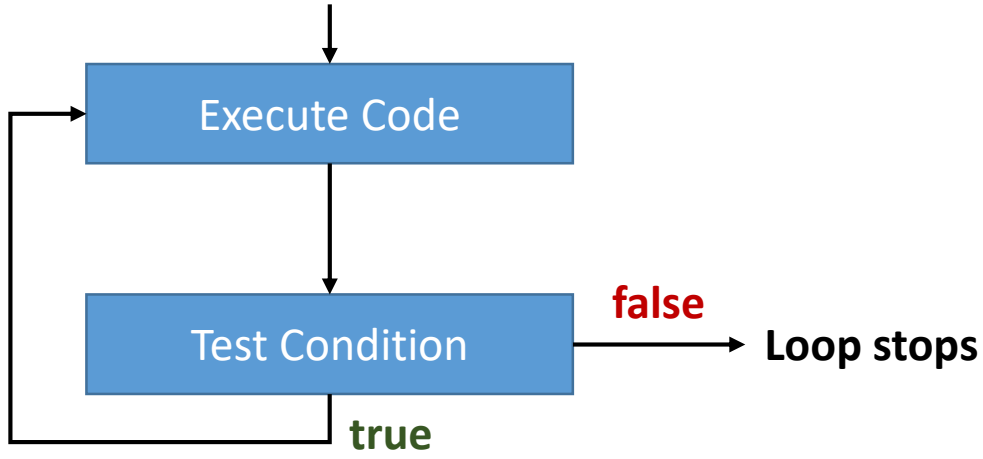
Repetitive Structures

- 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.

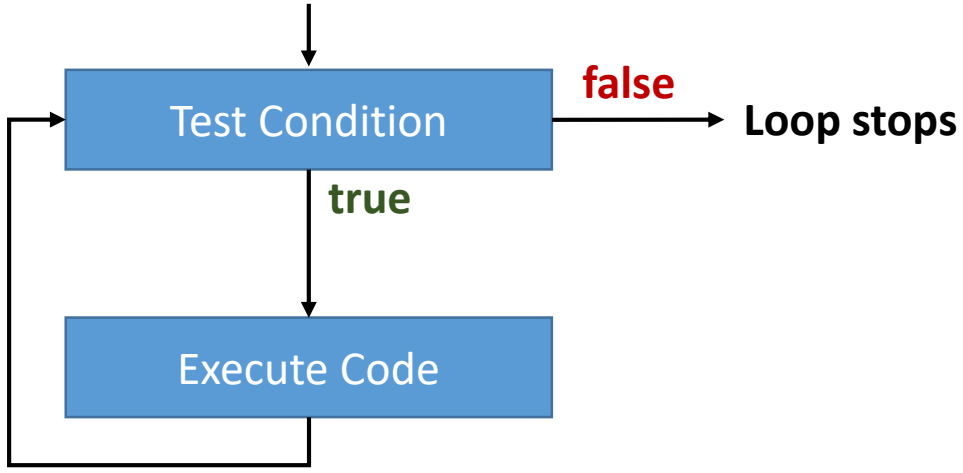
```
do {
   //Code that executes at least once
   //and iterates as long as the
   //condition is true
} while(Boolean expression);
```

Semicolon!

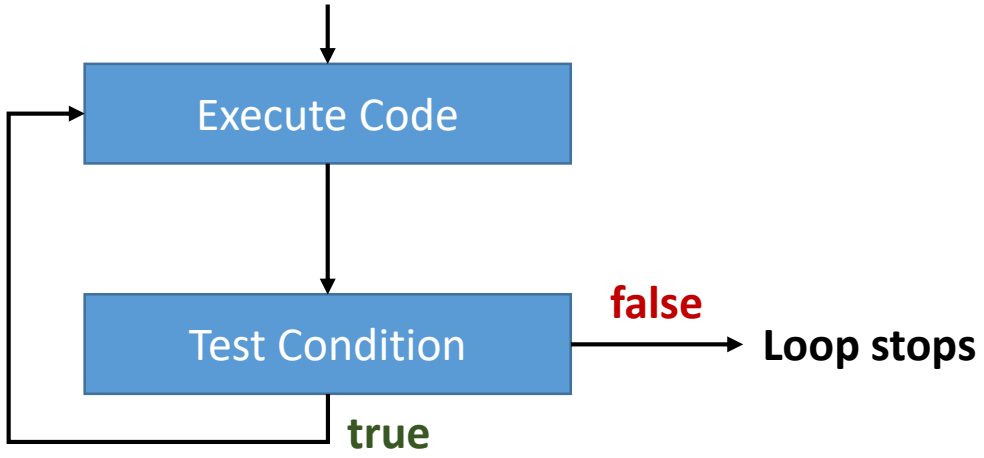
Repetitive Structures



Repetitive Structures



While Loop



Do-While Loop

Repetitive Structures

- 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.

Repetitive Structures

- 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.

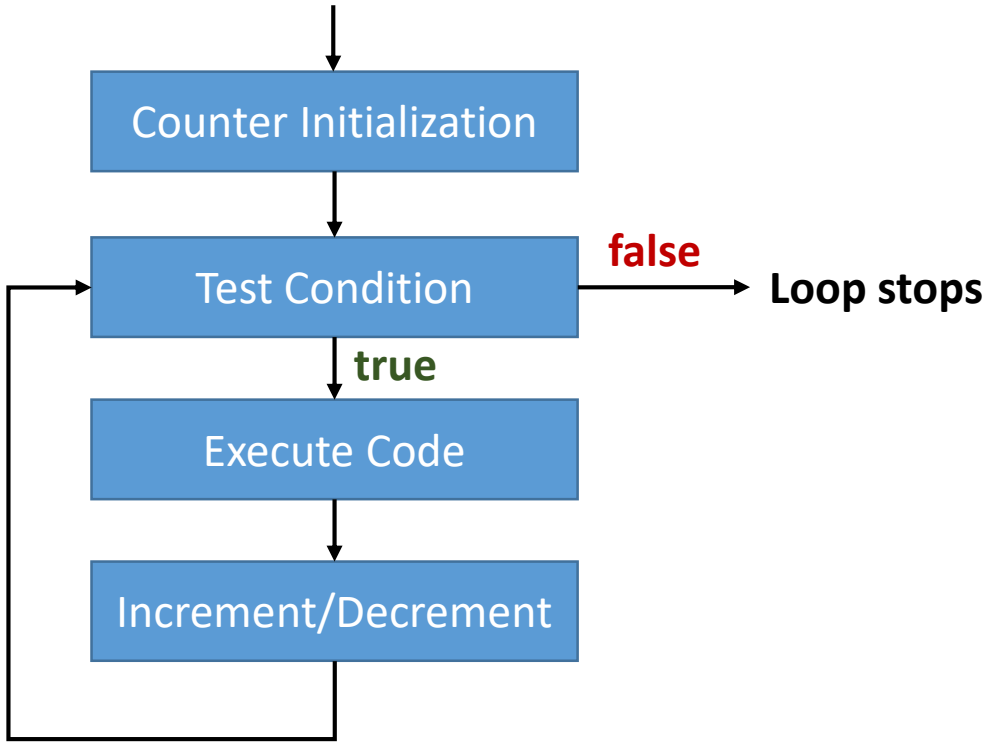
- Java's enhanced for loop will be demonstrated in a future lecture.

Repetitive Structures

- 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
}
```

Repetitive Structures



Repetitive Structures

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.

Repetitive Structures

```
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!
```

Repetitive Structures

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

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

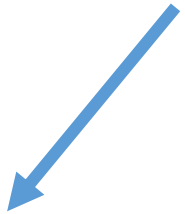
Repetitive Structures

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

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

Repetitive Structures

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
```

Repetitive Structures

- “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.

Repetitive Structures

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

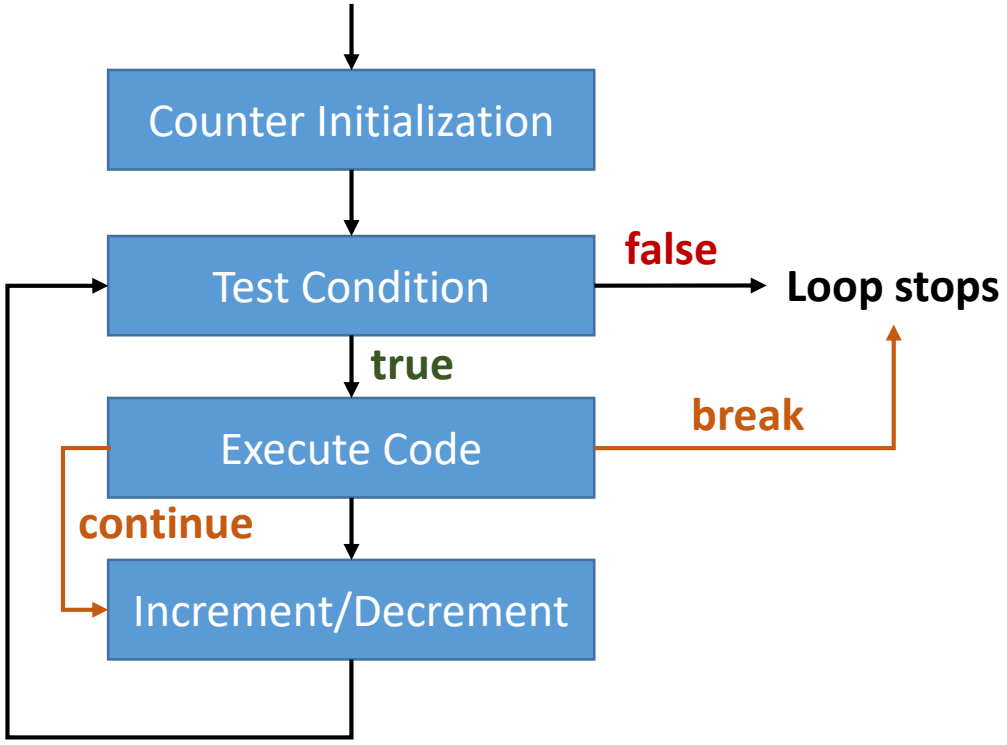
`break;`

- 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.

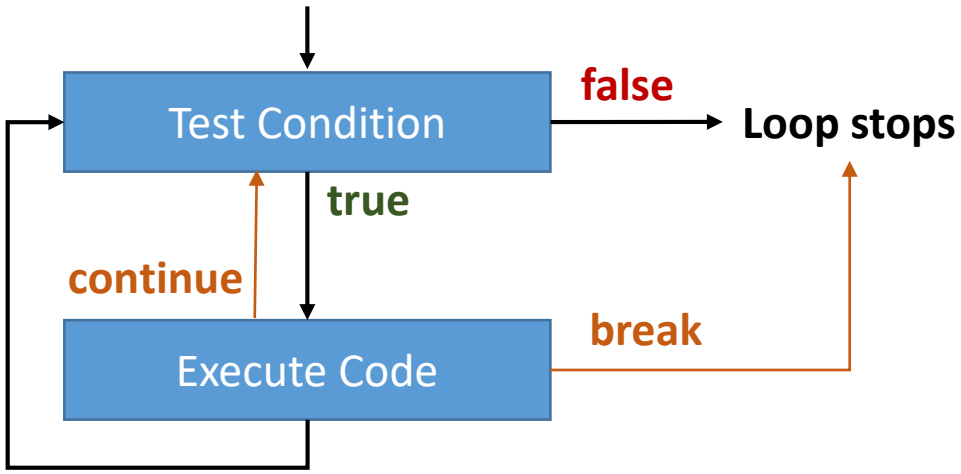
`continue;`

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

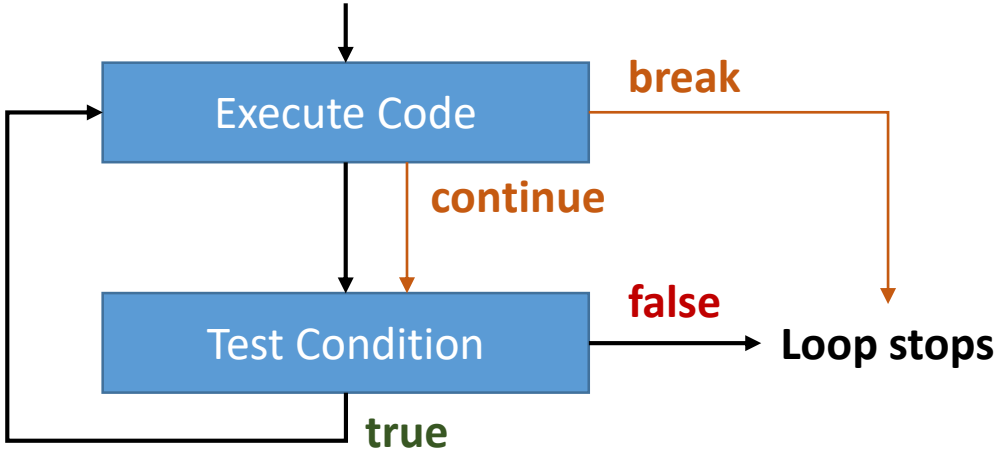
For Loop (Updated Flow Chart)



While & Do-While Loop (Flow Charts)



While Loop



Do-While Loop

Repetitive Structures

```
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.

Repetitive Structures

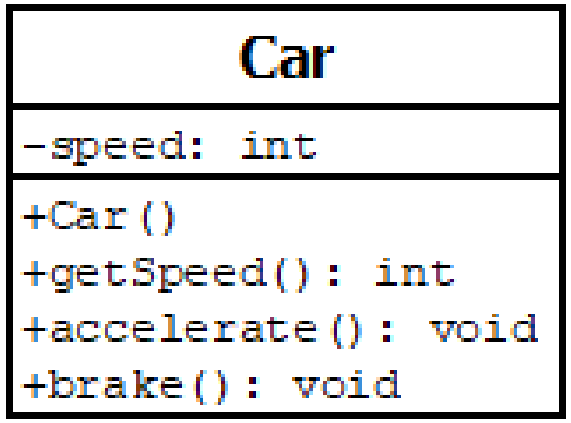
```
for(int myInt = 2; myInt <= 11; myInt++) {
  if(myInt % 2 == 1) {
    continue;
  }
  System.out.println("Number: " + myInt);
}
System.out.println("All done!");
```

```
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.

Repetitive Structures

```
public class Car {

     private int speed;

     public Car() {
          speed = 0;
     }

     public int getSpeed() {
          return speed;
     }

     public void accelerate() {
          speed += 5;
     }

     public void brake() {
          speed -= 5;
     }

}
```



- This example shows a class that models a Car.
  - There are *many* more attributes that could be added such as the color, make, model, year, etc.

Repetitive Structures

```
public class TestProgram {

    public static void main(String[] args) {
        Car testCar = new Car();
        System.out.println("Speed: " + testCar.getSpeed());

        for(int i = 0; i < 5; i++) {
            testCar.accelerate();
        }
        System.out.println("Speed: " + testCar.getSpeed());

        for(int i = 0; i < 3; i++) {
            testCar.brake();
        }
        System.out.println("Speed: " + testCar.getSpeed());
    }

}
```

```
Speed: 0
Speed: 25
Speed: 10
```

Aggregation

- Real-world objects are typically comprised of several other objects.
  - For example, a bicycle is made up of tires, a chain, pedals, handlebars, etc.

- A software object can be designed in a similar way.
  - The more complex objects contain and utilize (“*aggregate*”) the smaller, simpler objects.

- In object-oriented programming, **aggregation** is a type of encapsulation that creates a “***has a***” relationship between classes.
  - A bicycle “has” tires.
  - A car “has a” steering wheel.
  - A classroom “has a” whiteboard.

Aggregation

- The below example shows a Bicycle class.
  - There are many more attributes that could be added such as what gear the bike is in, the color it is, etc.

```
public class Bicycle {

	private int speed;

	public Bicycle() {
		speed = 0;
	}

	public int getSpeed() {
		return speed;
	}

	public void setSpeed(int s) {
		speed = s;
	}

}
```

Aggregation

- The below example shows a class for a Tire object.
  - There are more attributes that could be added such as radius, type of tread, etc.

```
public class Tire {

	private int pressure;

	public Tire(int p) {
		pressure = p;
	}

	public int getPressure() {
		return pressure;
	}

	public void setPressure(int p) {
		pressure = p;
	}

}
```

Aggregation

- The below example shows the Bicycle class with two new fields.
  - Both of which are Tire objects.

```
public class Bicycle {

	private int speed;
	private Tire front;
	private Tire back;

	public Bicycle() {
		speed = 0;
		front = new Tire(45);
		back = new Tire(45);
	}

	public int getSpeed() {
		return speed;
	}

	public void setSpeed(int s) {
		speed = s;
	}

}
```

Aggregation

```
public class Bicycle {

	private int speed;
	private Tire front;
	private Tire back;

	public Bicycle() {
		speed = 0;
		front = new Tire(45);
		back = new Tire(45);
	}

	public int getBackPressure() {
		return back.getPressure();
	}

	public void setBackPressure(int p) {
		back.setPressure(p);
	}

	public int getSpeed() {
		return speed;
	}

	public void setSpeed(int s) {
		speed = s;
	}

}
```

- Adding a setter and getter for the back tire's pressure
- Similar methods could be added for the front tire. For brevity, we'll work only with the back tire

Aggregation

```
public class TestProgram {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();

        System.out.println("Back Pressure: " + testBike.getBackPressure());
        testBike.setBackPressure(32);
        System.out.println("Back Pressure: " + testBike.getBackPressure());
    }

}
```

```
Back Pressure: 45
Back Pressure: 32
```

Aggregation

```
public void setBackPressure(int p) {
    back.setPressure(p);
    if(p < 1) {
        speed = 0;
    }
}

public void setSpeed(int s) {
    if(back.getPressure() > 0) {
        speed = s;
    }
    else {
        speed = 0;
    }
}
```

- Perhaps we want to set the speed to zero if the back tire's pressure is too low (i.e., the tire is flat.)
- We'll need to update both the setSpeed and setBackPressure methods

Aggregation

```
public class TestProgram {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();

        System.out.println("Speed: " + testBike.getSpeed());
        System.out.println("Back Pressure: " + testBike.getBackPressure());

        testBike.setSpeed(10);
        System.out.println("Speed: " + testBike.getSpeed());

        testBike.setBackPressure(0);     //Flat Tire
        System.out.println("Speed: " + testBike.getSpeed());

    }

}
```

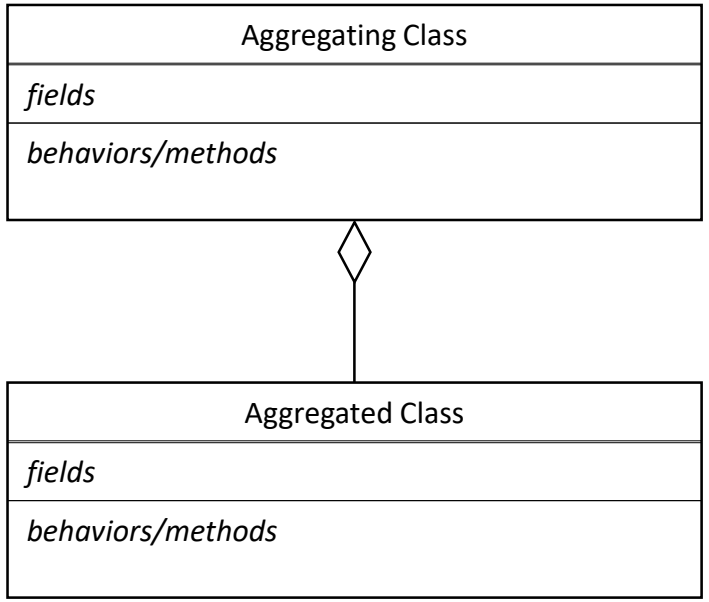
```
Speed: 0
Back Pressure: 45
Speed: 10
Speed: 0
```

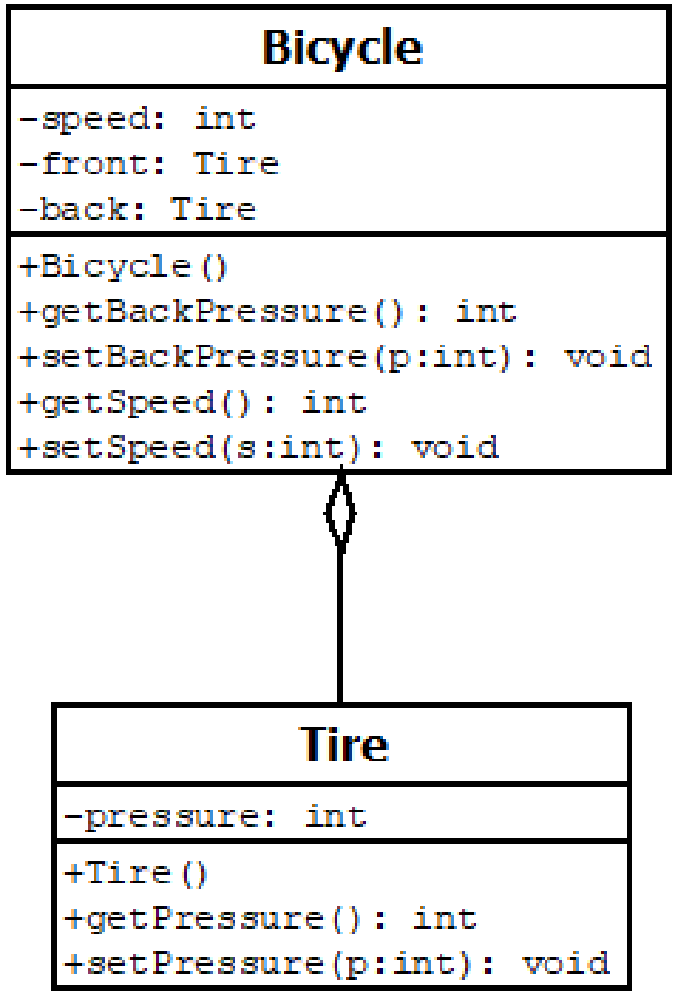
Aggregation

```
public class TestProgram {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();

        testBike.setBackPressure(0);     //Flat Tire
        System.out.println("Speed: " + testBike.getSpeed());

        testBike.setSpeed(100);
        System.out.println("Speed: " + testBike.getSpeed());

        testBike.setBackPressure(35);
        testBike.setSpeed(15);
        System.out.println("Speed: " + testBike.getSpeed());

    }

}
```

```
Speed: 0
Speed: 0
Speed: 15
```

Aggregation





Random Number Generators

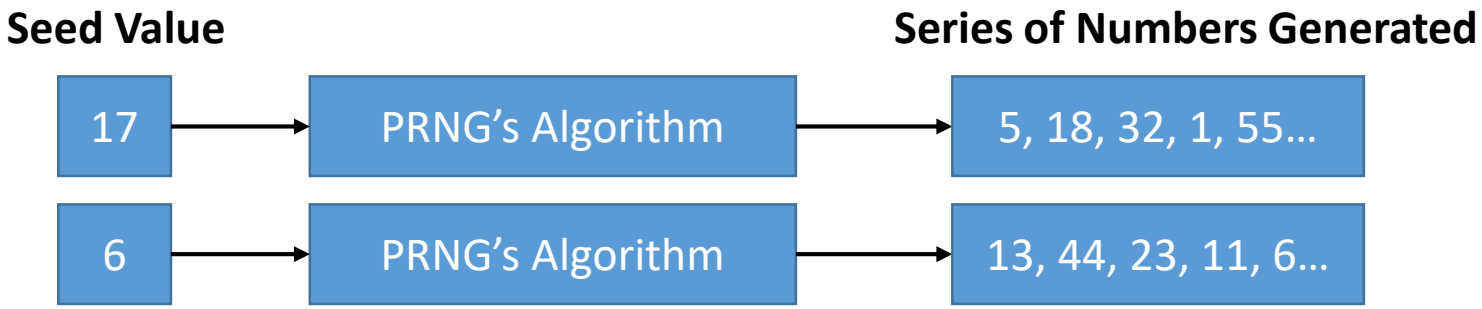
- A ***random number generator*** is software or hardware that produces a random number.
  - A ***random number*** is number chosen from a set of possible values, each with the same probability of being selected.

- 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

```
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);
    }

}
```

Reading and Writing Text Files

- The File object provides many methods that gives us information about a file.
  - It must be imported: **`import java.io.File;`**

- There are a number of constructors for a File object, but we'll be using the constructor with one String parameter- the path to the file.

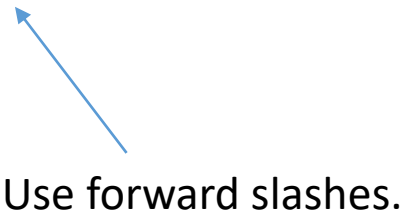
```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
```

\ indicates the start of an escape sequence.
Need to use \\ in Strings containing a file's path.

Reading and Writing Text Files

- MAC AND LINUX USERS:

```
File myTextFile = new File("/path/to/my/file.txt");
```



Reading and Writing Text Files

- Pass your file to a new Scanner object.
  - This is the same Scanner object you have been using to get keyboard input.
  - Now, we are using the file as the input stream instead of System.in

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);
```

Reading and Writing Text Files

- The nextLine method will return the next line of the file as a String.
  - In this example, it would return line 1 from file.txt, since this is the first time we called the nextLine method.
- This only reads a single line. How can we read through an entire file with an unknown number of lines?

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);
String line = fileReader.nextLine();
```

Reading and Writing Text Files

- The Scanner's hasNextLine method returns true if there are more lines to be read and false if it reached the end of the file.
- The below while loop will iterate as long as there are still lines to be read.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);

while(fileReader.hasNextLine()) {
    System.out.println(fileReader.nextLine());
}
```

Reading and Writing Text Files

- The Scanner's close method releases it's hold on the file.
- Only call this method when you are done using the resource/file.

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
Scanner fileReader = new Scanner(myTextFile);

while(fileReader.hasNextLine()) {
    System.out.println(fileReader.nextLine());
}

fileReader.close();
```

Reading and Writing Text Files

- The PrintWriter object provides an easy way to write data to a new or existing file.
  - Must be imported: **`import java.io.PrintWriter;`**

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
PrintWriter fileWriter = new PrintWriter(myTextFile);
```

- The above example initializes a new PrintWriter object with a File object.
  - The file the PrintWriter will write to.

Reading and Writing Text Files

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
PrintWriter fileWriter = new PrintWriter(myTextFile);
fileWriter.println("This will be printed to my file.");
```

- The PrintWriter's **println**, **print**, and **printf** methods work in an identical fashion to System.out.println/print/printf.
  - The "out" in System.out *is* a PrintWriter. It just prints to the console/"standard output" instead of to a file.

Reading and Writing Text Files

```
File myTextFile = new File("C:\\path\\to\\my\\file.txt");
PrintWriter fileWriter = new PrintWriter(myTextFile);
fileWriter.println("This will be printed to my file.");
fileWriter.print("This will also be printed to my file.");
fileWriter.println("And so will this.");
fileWriter.close();
```

- If you do not close your PrintWriter after you are finished writing to the file, your changes will **not** be made permanent and the file will be empty.

Reading and Writing Text Files

```
import java.io.BufferedWriter;
import java.io.FileWriter;
```

```
PrintWriter fileWriter = new PrintWriter(new BufferedWriter(new FileWriter("OutputFile.txt", true)));
fileWriter.println("This will be appended to the existing file.");
fileWriter.close();
```

- A few more steps than using just PrintWriter and File objects.
- The above will append the line “This will be appended to the existing file.” to the end of OutputFile.txt instead of completely overwriting its existing data.
- This is just for your reference.

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

```
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);
```