Object-Oriented Programming III

Michael C. Hackett

Assistant Professor, Computer Science

Community
College
of Philadelphia

Lecture Topics

- Access Modifiers
- Accessor Methods
- Mutator Methods
- Relational and Logical Operators
- Decision Structures
- Variable Scope
- Method Overloading
- Encapsulation

- Extras
  - String Comparison
  - Switch Structures
  - Inline If Statements

Access Modifiers

- **Access modifiers** specify how classes, fields, and methods can be accessed by other objects.
  - This limits other objects from making changes to the data in the fields or calling methods that pertain only to the object's internal implementation.

public

- Allows a field or method to be accessible to all other objects.

private

- Does not allow a field or method to be accessible to other objects.

Access Modifiers

- When there is no access modifier present (like in the examples shown thus far) then the class, field, or method is said to be **package private**.
  - Other classes within the same package may access that class, field and method; classes outside of that package cannot.

- From this point forward, we will use an access modifier for all classes, fields, constructors, and methods.

Access Modifiers

```
public class Sphere {


}
```

- In most cases, a class will be either public or package private.

- Occasionally, a class might be private in the case of an inner class.
  - Inner classes will be one of the last topics we cover in this course.
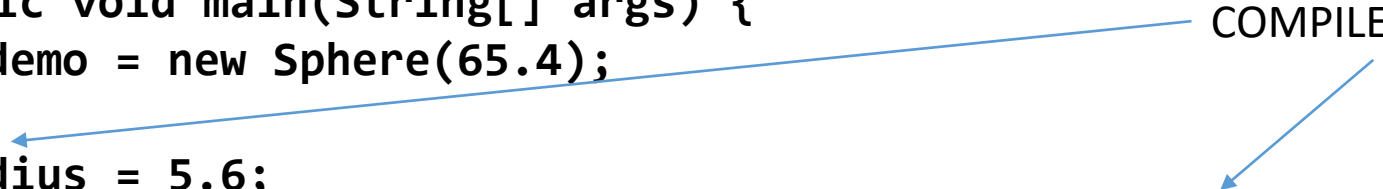
Access Modifiers

```
public class Sphere {

    private  double radius;

    public  Sphere(double r) {
        radius = r;
    }
}
```

- Normally, the fields of a class will be private.
  - This better encapsulates the object's data.
  - Only the object itself can change the value of a private field.

- Constructors are typically public, as we want other objects to be able to instantiate objects of this class.
  - Constructors are sometimes private in situations where inheritance is involved (not covered until CSCI 112)

Access Modifiers

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(65.4);

        demo.radius = 5.6;
        System.out.println("The sphere's radius is " + demo.radius);
    }
}
```
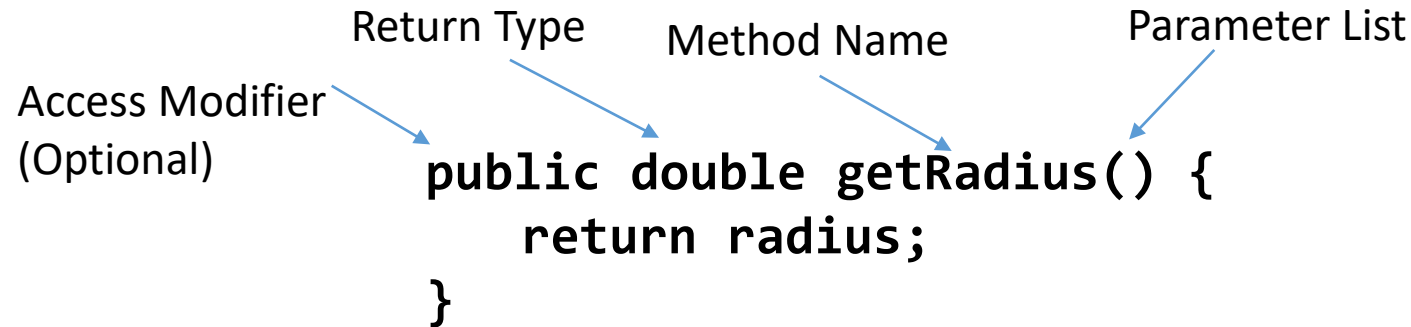
COMPILE ERROR

- A private field is not accessible outside of the class.
- The Sphere class can contain methods that either retrieve or change the value of a private field.

Accessor Methods

- An **accessor method** retrieves (or "gets") data from an object.
  - Colloquially called a "getter" method.

Return Type   Method Name   Parameter List

Access Modifier
(Optional)

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

- This method will return the current value of our radius field.
- The type of data returned must match the method's return type.
  - Since this method's return type is double, the method can only return a double value.

Accessor Methods

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }

}
```

Accessor Methods

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(65.4);

        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
The sphere's radius is 65.4

Mutator Methods

- A **mutator method** alters (or "sets") the data of an object.
  - Colloquially called a "setter" method.

Return Type   Method Name   Parameter List

Access Modifier
(Optional)

```
public void setRadius(double r) {
    radius = r;
}
```

- This method will change the current value of the object's radius field.
- A method that does not return data (no return statement) must have a **void** return type.
  - Mutators typically do not return data. They "set" data- they don't "get" data.

Mutator Methods

```java
public class Sphere {

    private double radius;

    public Sphere(double r) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        radius = r;
    }

}
```

Mutator Methods

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(65.4);

        System.out.println("The sphere's radius is " + demo.getRadius());

        demo.setRadius(104.5);
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
The sphere's radius is 65.4
The sphere's radius is 104.5

Python and Java Relational Operations

Purpose	Python	Java
Equality	`x = y == z`	`x = y == z;`
Inequality	`x = y != z`	`x = y != z;`
Less Than	`x = y < z`	`x = y < z;`
Greater Than	`x = y > z`	`x = y > z;`
Less Than or Equal To	`x = y <= z`	`x = y <= z;`
Greater Than or Equal To	`x = y >= z`	`x = y >= z;`

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

Python and Java Logical Operations

Purpose	Python	Java
And	`x = y and z`	`x = y && z;`
Or	`x = y or z`	`x = y || z;`
Not	`x = not y`	`x = !y;`

- Java's logical operators behave like Python's, but they are characters 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 (Negative sign)
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

Decision Structures

- 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
}
```

Decision Structures

```
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.
```

Decision Structures

- An else clause in Java works like an else clause in Python.

- The syntax for an else clause in Java is shown below.

```java
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
}
```

Decision Structures

```
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.
```

Decision Structures

```
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.

Decision Structures

- 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
}
```

Decision Structures

```
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.

Decision Structures

- 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.
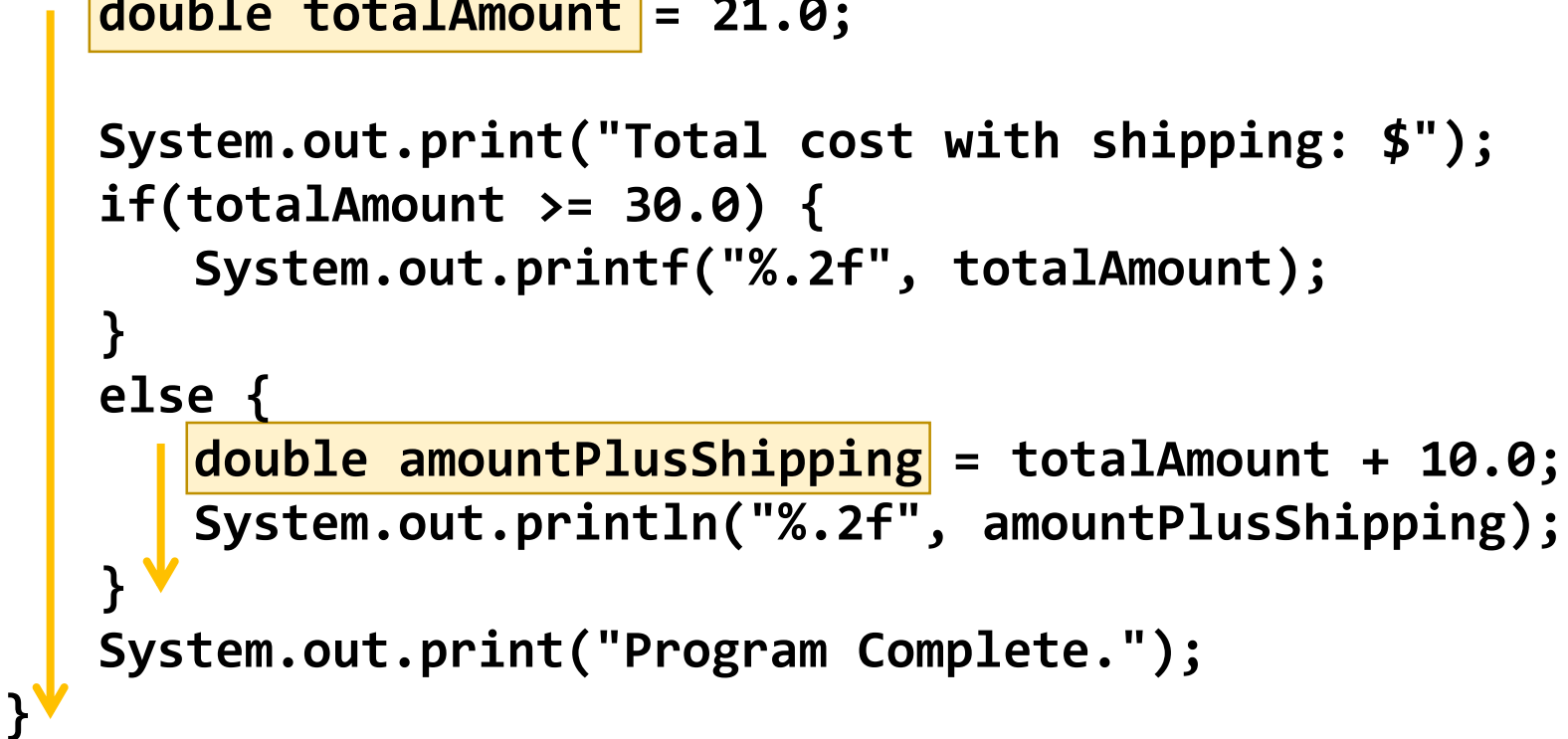
Variable Scope

```
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

```
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.");
}
```

Method Overloading

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        radius = r;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        radius = r;
    }

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

}
```

- Like constructors, methods can also be overloaded.

- Like overloaded constructors, overloaded methods must have unique signatures.
  - setRadius(double)
  - setRadius(String)

Method Overloading

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(65.4);

        System.out.println("The sphere's radius is " + demo.getRadius());

        demo.setRadius(14.5);
        System.out.println("The sphere's radius is " + demo.getRadius());

        demo.setRadius("83.214");
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
```
The sphere's radius is 65.4
The sphere's radius is 14.5
The sphere's radius is 83.214
```

Encapsulation

- Limiting access to an object's fields gives the object greater control over its encapsulated data.
    - Consider the example below, where the Sphere constructor allows any value (even values that don't make sense, like a negative value) to be set to the radius field

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
The sphere's radius is -73.65

Encapsulation

```
public class Sphere {

    private double radius;

    public Sphere(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

    ... (Other methods)
}
```

- The constructor now checks that the parameter's value is greater than 0
  - If it is, it assigns that value to the radius field
  - If it is not (zero or negative), it assigns a default value of 1 to the radius field.

Encapsulation

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
The sphere's radius is 1

Encapsulation

- While the constructor performs a check, the setter methods both still allow invalid data to be assigned to the radius field

```
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere's radius is " + demo.getRadius());
        demo.setRadius(-9.25);
        System.out.println("The sphere's radius is " + demo.getRadius());
        demo.setRadius("-200.13");
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
```
The sphere's radius is 1
The sphere's radius is -9.25
The sphere's radius is -200.13
```

Encapsulation

```java
public class Sphere {

    private double radius;

    public Sphere(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double r) {
        if(r > 0) {
            radius = r;
        }
        else {
            radius = 1;
        }
    }

    public void setRadius(String r) {
        if(Double.parseDouble(r) > 0) {
            radius = Double.parseDouble(r);
        }
        else {
            radius = 1;
        }
    }
}
```

- We could add the same logic to the two methods, but this adds:
  - More ways for the software to break
    - Adding a new setter method or constructor would depend on the programmer remembering to add this check
  - More source code
    - A benefit of object-oriented programming is that it is supposed to reduce the amount of source code
  - More of the *same* source code
    - A benefit of object-oriented programming is that it is supposed make existing code more reusable

Encapsulation

```
public class Sphere {

    private double radius;

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

    public double getRadius() {
        return radius;
    }

    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;
        }
    }
}
```

- A solution is to create a utility method that specifically handles validating the data to be assigned to the radius field:
  - The method is private, so that only the other methods in the Sphere class can call on it
  - This one method can contain all necessary logic for error-checking data prior to assigning anything to the object's radius field.
  - Any new methods or constructors in class can simply call this method and it will ensure that the radius field is set correctly.

Encapsulation

```java
public class TestProgram {

    public static void main(String[] args) {
        Sphere demo = new Sphere(-73.65);
        System.out.println("The sphere's radius is " + demo.getRadius());
        demo.setRadius(-9.25);
        System.out.println("The sphere's radius is " + demo.getRadius());
        demo.setRadius("-200.13");
        System.out.println("The sphere's radius is " + demo.getRadius());
    }
}
```

Output
```
The sphere's radius is 1
The sphere's radius is 1
The sphere's radius is 1
```

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";
```

catString1

catString2

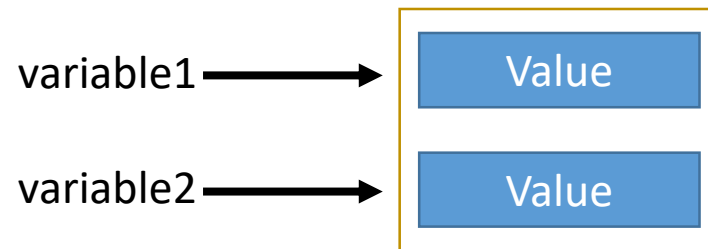dogString1

String Pool

Cats

Dogs
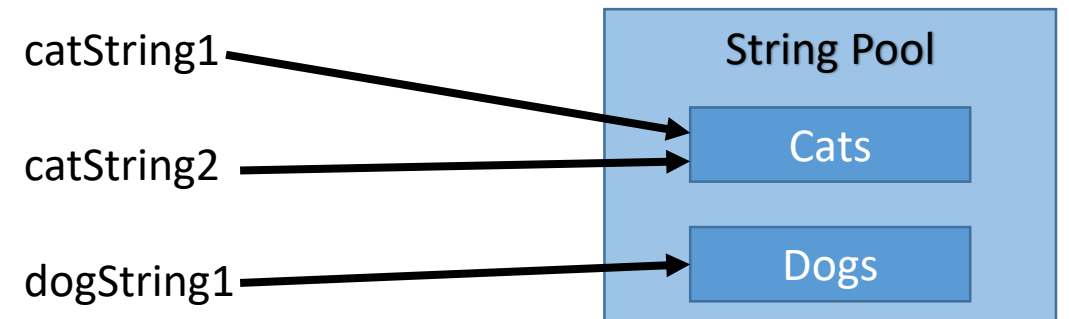
String Comparison

- There are two ways to test the equality of Strings:

  1. Testing if two String variables *reference* the same value in memory.
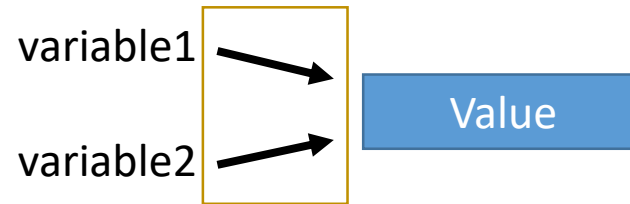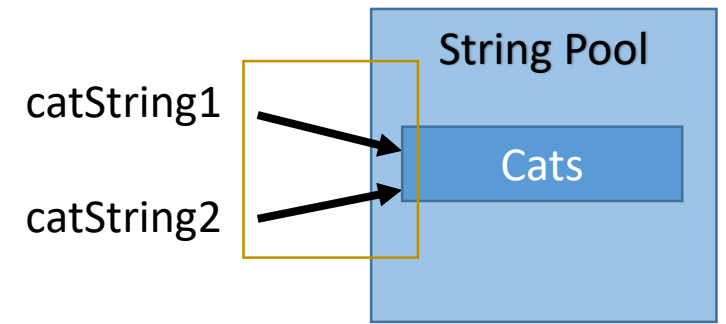

  2. Testing if the *values* of two String variables are the same.

String Comparison (Reference)

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

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

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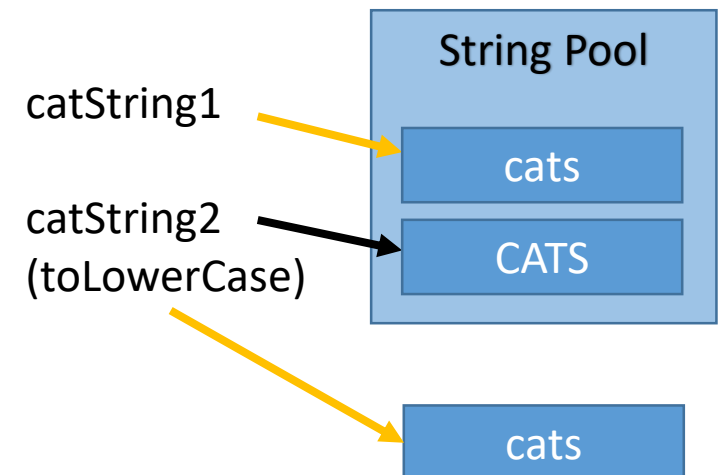
String Comparison (Reference)

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

                 false
if(catString1 == catString2.toLowerCase()) {
    System.out.println("The Strings are equal");
}
```
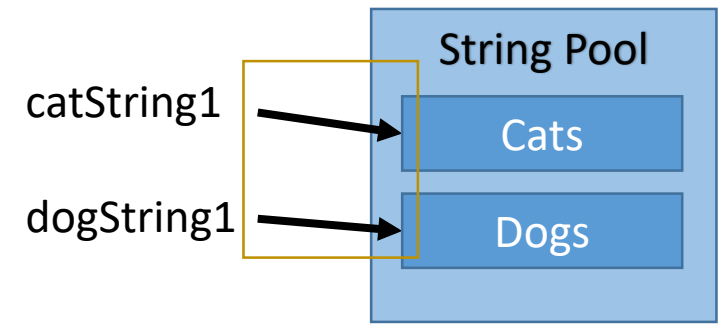
- The references are compared, not the values.

String Pool

catString1 → cats

catString2
(toLowerCase) → CATS

cats

String Comparison (Reference)

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

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

String Pool

catString1 → Cats

dogString1 → Dogs

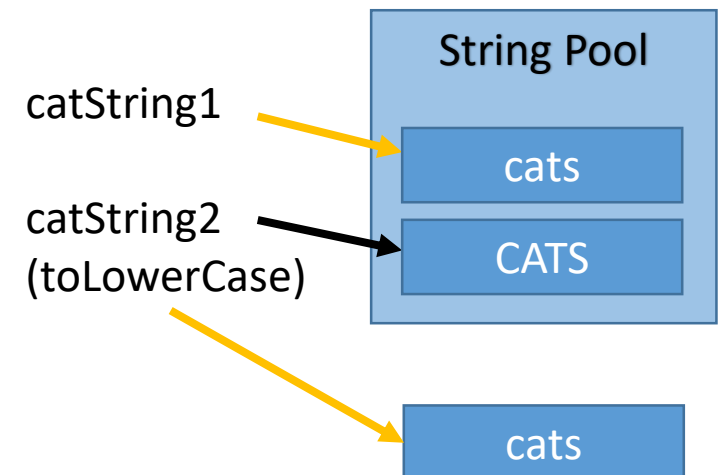# String Comparison (Reference)

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

          true
if(catString1 != catString2.toLowerCase()) {
    System.out.println("The Strings are not equal");
}
```
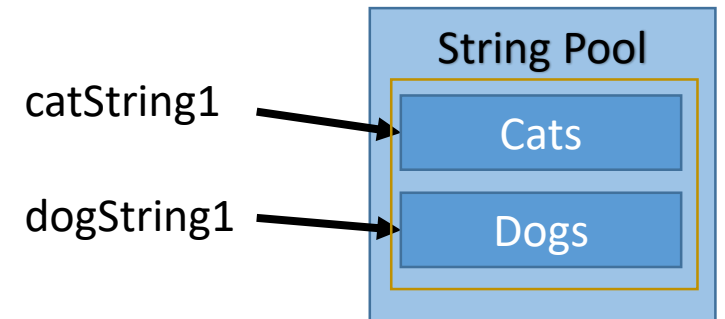
- The references are compared, not the values.

String Pool

catString1 → cats

catString2
(toLowerCase) → CATS

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 Pool

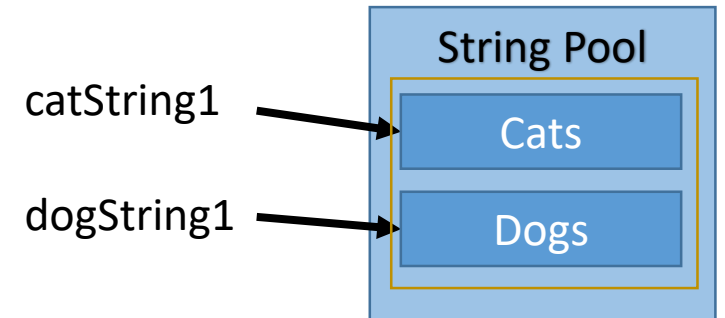catString1

Cats

dogString1

Dogs

String Comparison (Value)

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

```
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");
}
```

catString1

dogString1

String Pool

Cats

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");
}
```

catString1

catString2

String Pool

Cats

CATS
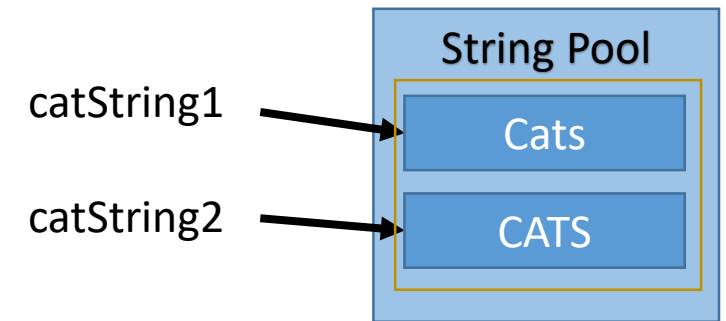
String Comparison (Length)

- 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)

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

                  false
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

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

```
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.

```
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.");
}
```

```
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

```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.");
             break;
    default: System.out.print("This is ");
             System.out.println("the default.");
}
```

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

Switch Structures (Strings)

```java
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.

```
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.");
}
```

```
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

```java
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

```
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;
```

Equivalent to

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

```
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;


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

Removed

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

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

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

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

pa