CSE 1325

Week of 09/26/2022

Instructor : Donna French

Conditional Operator

- Conditional operator can be used in place of certain types of `if-else` statements

- ternary operator
  - takes 3 operands
  - boolean expression ? value if true : value if false

```
Random rn = new Random();
int RandomNumber = rn.nextI
String status;

if (RandomNumber % 2 == 0)
{
      status = "even";
}
else
{
      status = "odd";
}
System.out.printf("Random number %d is %s\n", RandomNumber, status);

Random number 6 is even
Random number 2 is even
Random number 5 is odd
Random number 0 is even
```

What range of random numbers will this produce?
0-9

```
Random rn = new Random();
int RandomNumber = rn.nextInt(10);
String status;

if (RandomNumber % 2 == 0)
{
      status = "even";
}
else
{
      status = "odd";
}
System.out.printf("Random number %d is %s\n", RandomNumber, status);
```

boolean expression ? value if true : value if false

```
Random rn = new Random();
int RandomNumber = rn.nextInt(10);

System.out.printf("Random number %d is %s\n", RandomNumber,
     (RandomNumber % 2 == 0) ? "even" : "odd");
```

enhanced for loop

It is common to process *all* the elements of an `ArrayList`.

The enhanced for loop allows you to do this *without using a counter*.

This statement avoids the possibility of "stepping outside" the `ArrayList` and eliminating the need for bounds checking.

When processing all elements of an `ArrayList`, if you do not need to access to an `ArrayList` element's subscript, use the enhanced for loop.

enhanced for loop

```
ArrayList<Integer> MyList = new ArrayList<>();

for (int i = 34; i < 52; i++)
{
    MyList.add(i*23);
}
```

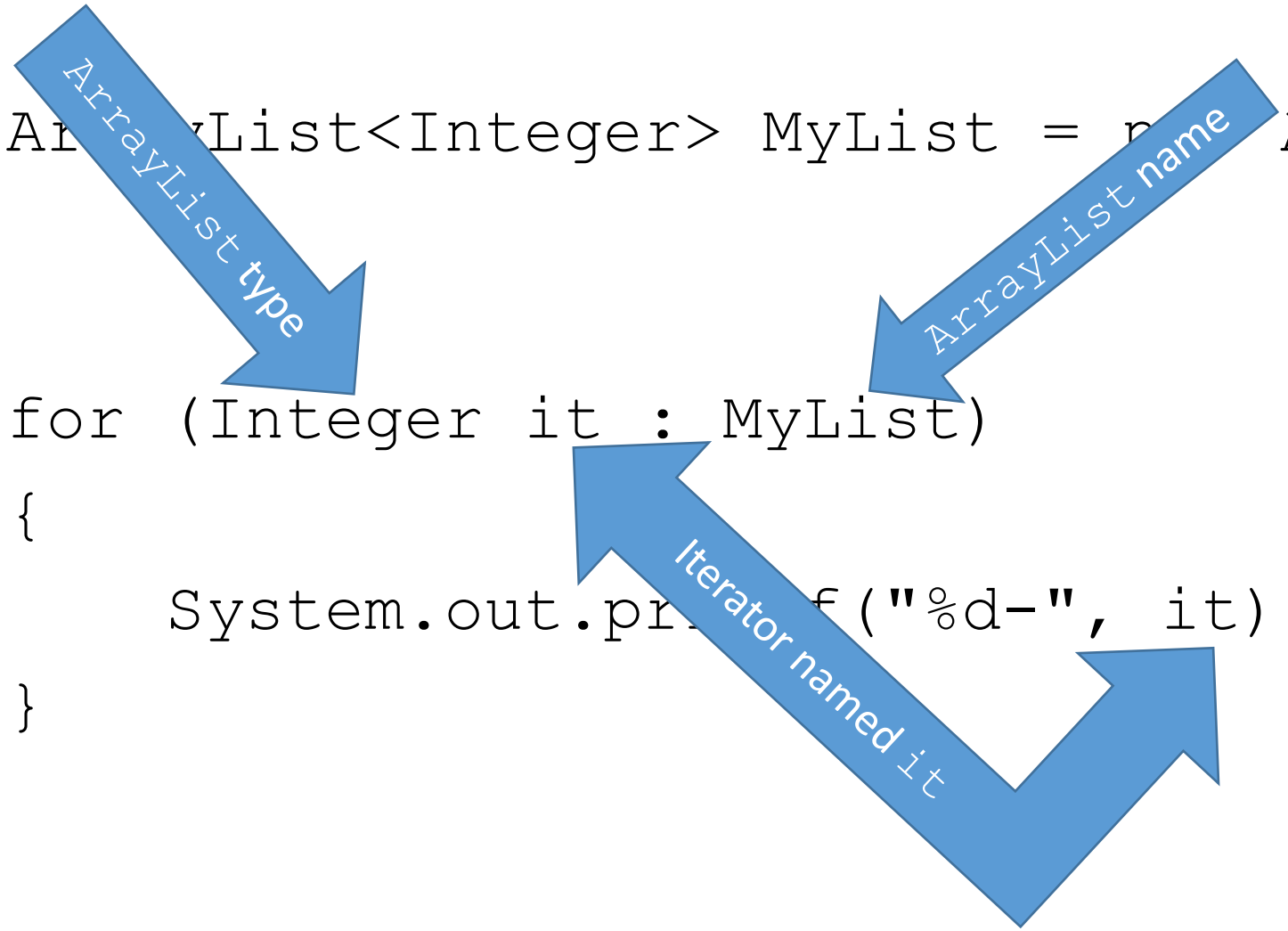
enhanced for loop

```
for (int i = 0; i < MyList.size(); i++)
{
    System.out.printf("%d-", MyList.get(i));
}
```

```
782-805-828-851-874-897-920-943-966-989-1012-1035-1058-1081-1104-1127-1150-1173-
```

```
for (Integer it : MyList)
{
    System.out.printf("%d-", it);
}
```

enhanced for loop




```
ArrayList<Integer> MyList = new ArrayList<>();


for (Integer it : MyList)
{
    System.out.printf("%d-", it);
}
```

for each iteration, assign the next element of `MyList` to `Integer` iterator `it`, then execute the loop body

```
ArrayList<String> aBunch = new ArrayList<>();

aBunch.add("Cavendish");
aBunch.add("Pisang Raja");
aBunch.add("Blue Java");
aBunch.add("Manzano");

for (String banana : aBunch)
{
    System.out.print(banana);
    System.out.printf("\n==============\n");
}
```

ArrayList type?

String

Iterator name?

banana

ArrayList name?

aBunch

`ArrayList`

The enhanced for loop can be used in place of the counter-controlled for loop whenever the code looping over an `ArrayList` does not require access to the element's subscript.

If a program needs use subscripts for some reason other than simply to loop through the `ArrayList`

to print a subscript number next to each array element value

then use the counter-controlled for statement.

ArrayList

```
ArrayList<Double> Grades = new ArrayList<>();
double Sum = 0;

File FH = new File("Grades.txt");
Scanner inFile = new Scanner(FH);

while (inFile.hasNextDouble())
{
    Grades.add(inFile.nextDouble());
}
```

```
ArrayList<Double> Grades = new ArrayList<>();
double Sum = 0;

File FH = new File("Grades.txt");
try
{
    Scanner inFile = new Scanner(FH);

    while (inFile.hasNextDouble())
    {
        Grades.add(inFile.nextDouble());
    }
}
catch (Exception frog)
{
    System.out.println("File did not open...exiting");
    System.exit(0);
}
```

```
ArrayList<Double> Grades = new ArrayList<>();
double Sum = 0;
Scanner inFile = null;

File FH = new File("Grades.txt");
try
{
    inFile = new Scanner(FH);
}
catch (Exception frog)
{
    System.out.println("File did not open...exiting");
    System.exit(0);
}

while (inFile.hasNextDouble())
{
    Grades.add(inFile.nextDouble());
}
```

ArrayList

Change to an enhanced for loop

```
for (int i = 0; i < Grades.size(); i++)
{
    Sum += Grades.get(i);
}
```

```
for (_Double_ _aGrade_: _Grades_)
{
    Sum += aGrade;
}
```

`ArrayList` type?
`Double`

Iterator name?
`aGrade`

`ArrayList` name?
`Grades`

Methods

Methods facilitate the design, implementation, operation and maintenance of large programs.

Normally, methods are called on specific objects

```
Random rn = new Random();
int MyRandomInt = rn.nextInt();
```

`static` methods can be called on a class rather than an object.

```
double MyRandomDouble = Math.random();
```

Methods

Sometimes a method performs a task that does not depend on the contents of any object.

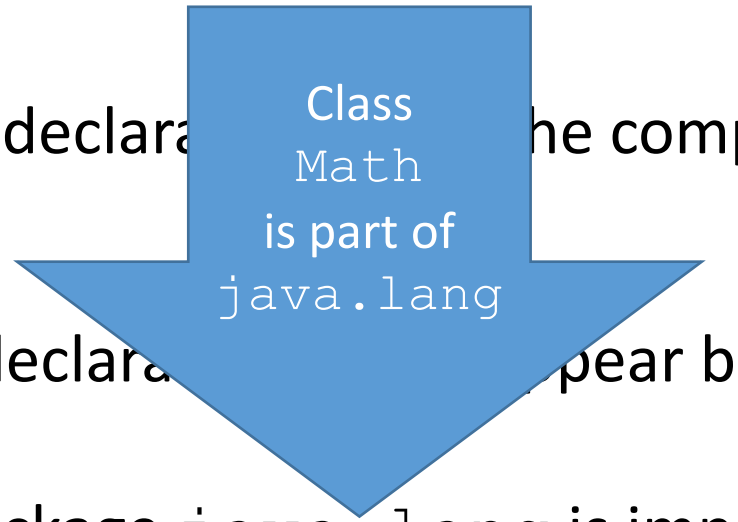
Method applies to the class in which it's declared as a whole and is known as a `static` method or a class method

For any class imported into your program, you can call the class's `static` methods by specifying the name of the class in which the method is declared, followed by a dot (.) and the method name

Method arguments may be constants, variables or expressions

import

- As we have seen, Java has a lot of predefined classes that we can use over and over
- These classes are grouped into packages
  - named groups of related classes
  - Java class library or Java Application Programming Interface (API)
- The `import` declara… …he compiler locate a class when you use it in your program
- All `import` declara… …pear before the first class definition in the file
- By default, package `java.lang` is imported in every Java program
  - classes in `java.lang` are the only ones in the Java API that do not require an `import`



methods

Class `Math` declares two constants

- `Math.PI` (`3.141592653589793`) is the ratio of a circle's circumference to its diameter
- `Math.E` (`2.718281828459045`) is the base value for natural logarithms

These constants are declared in class `Math` with the modifiers

`public`
: allows you to use these fields in your own classes

`static`
: allows them to be accessed via the class name `Math` and a dot (.) separator

`final`
: indicates that they are constants—value cannot change after they are initialized.

A class's variables are sometimes called fields.

java.lang.Math

```
87:    /**
88:     * The most accurate approximation to the mathematical constant <em>e</em>:
89:     * <code>2.718281828459045</code>. Used in natural log and exp.
90:     *
91:     * @see #log(double)
92:     * @see #exp(double)
93:     */
94:    public static final double E = 2.718281828459045;
95:
96:    /**
97:     * The most accurate approximation to the mathematical constant <em>pi</em>:
98:     * <code>3.141592653589793</code>. This is the ratio of a circle's diameter
99:     * to its circumference.
00:     */
101:    public static final double PI = 3.141592653589793;
```

Why is method `main()` declared `static`?

When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify

Declaring `main` as `static` allows the JVM to invoke `main` without creating an object of the class

```
public class Code2_1000074079
```

```
javac Code2_1000074079.java
```

methods

A `public` method is “available to the public”

- Can be called from methods of other classes

`static` methods in the same class can call each other directly

- Any other class that uses a `static` method must fully qualify the method name with the class name

For now, we begin every method declaration with the keywords `public` and `static`

methods

Three ways to call a method

Using a method name by itself to call another method of the same class

```
int menu_choice = PencilMenu();
```

Using an object's variable name followed by a dot (.) and the method name to call a non-`static` method of the object

```
File FH = new File("input.txt");
if (FH.exists())
```

Using the class name and a dot (.) to call a static method of a class

```
double answer = Math.pow(2,76);
```

Method Overloading

Method overloading is a feature of Java that allows us to create multiple functions with the same name, so long as they have different parameters.

Consider this function…

```
public static int funA(int X, int Y, int Z)
{
  return X+Y+Z;
}
```

Method Overloading

```
public static int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}

public static void main(String[] args)
{
    System.out.println(funA(2,2,2));
    System.out.println(funA(3.3,3.3,3.3));
}
```

error: incompatible types: possible lossy conversion from double to int

Method Overloading

What if we created a method with the same name but used a different type for the parameters?

```
public static int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}
```

Prints 6

```
public static double funA(double X, double Y, double Z)
{
    return X+Y+Z;
}
```

Prints 9.899999999999999

Method Overloading

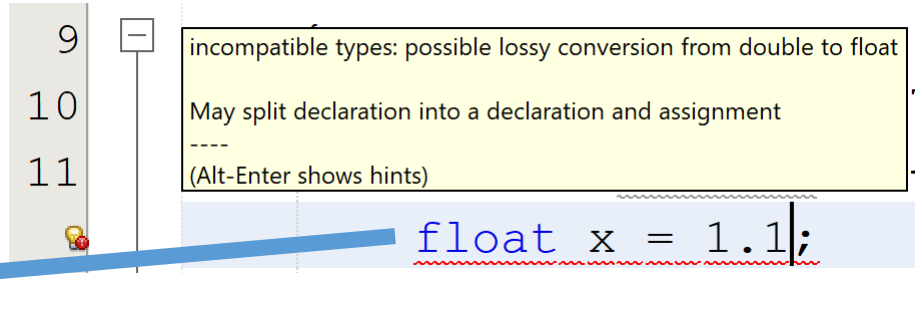
Method overloading is a feature of Java that allows us to create multiple functions with the same name, so long as they have different parameters.

Method overloading is used to create several methods that perform the same or similar tasks on different types or different numbers of arguments.

We have already seen this…

Method Overloading

```
int iNumber = -987;
double dNumber = -9.87;
float fNumber = -9.984F;
long lNumber = -12345677;
```




```
System.out.println(Math.abs(iNumber));      987
System.out.println(Math.abs(dNumber));      9.87
System.out.println(Math.abs(fNumber));      9.984
System.out.println(Math.abs(lNumber));      12345677
```

Method Overloading

The compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.

The combination of the method's name and the number, types and order of its parameters, but not its return type.

Method calls cannot be distinguished by return type

- Overloaded methods can have different return types if the methods have different parameter lists

Overloaded methods need not have the same number of parameters

Method Overloading

```
public static int funA(int X, int Y,
{
  return X+Y+Z;
}

public static double funA(double X, double Y, double Z)
{
  return X+Y+Z;
}

public static void main(String[] args)
{
    System.out.println(funA(2,2,2));
    System.out.println(funA(3.3,3.3,3.3));
}
```

signature – funA+int+int+int

signature – funA+double+double+double

Method Overloading

```
public static int funA(int X, int Y, int Z)
{
    return X+Y+Z;
}

public static double funA(double X, double Y, double Z)
{
    return X+Y+Z;
}

public static void main(String[] args)
{
    System.out.println(funA(2, 2, 2));
    System.out.println(funA(3.3, 3.3, 3.3));
    System.out.println(funA(2, 2.2, 2));
}
```

```
6
9.899999999999999
6.2
```

Method Overloading

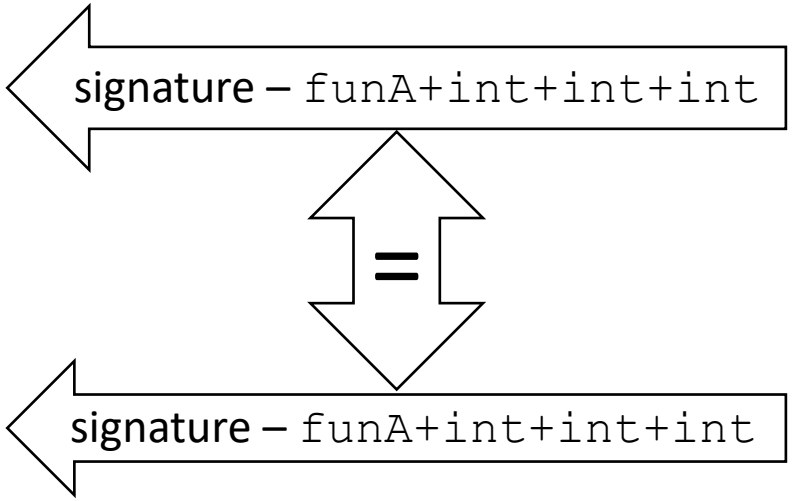
What is the signature of each of these methods?

```
public static String displayMoney(int amount)

      signature = displayMoney + int

public static ACTION buyPencils(final int PENCIL_PRICE, int payment,
                                int quantity, int Levels[], String change[])

      signature = buyPencils+final int + int + int + int[] + String[]

public static int PencilMenu()

      signature = PencilMenu
```

Method Overloading

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

```
public static int funA(int X, int Y, int Z)
{
  return X*Y*Z;
}
public static double funA(int X, int Y, int Z)
{
  return X*Y*Z;
}
```



error: method funA(int,int,int) is already defined in class MethodOverloading

Method Overloading

Method overloading can lower a program’s complexity significantly while introducing very little additional risk.

Method overloading typically works transparently and without any issues.

The compiler will flag any methods with duplicate signatures which can then be resolved.

Conclusion : Method overloading can make your program simpler.

Variable Length Argument Lists

- You can create methods that receive an unspecified number of arguments using variable-length argument lists.
- A type followed by an ellipsis (...) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type.
- An ellipsis can occur only once in a parameter list and the ellipsis, together with its type and the parameter name, must be placed at the end of the parameter list.

Variable Length Argument Lists

```
public static void main(String[] args)
{
   double d1 = 12.3;
   double d2 = 45.6;
   double d3 = 78.9;
   double d4 = 14.7;

   System.out.printf("d1 = %.2f\nd2 = %.2f\nd3 = %.2f\nd4 = %.2f\n", d1,d2,d3,d4);

   System.out.printf("Average of d1 and d2 = %.2f\n", average(d1, d2));
   System.out.printf("Average of d1 and d2 and d3 = %.2f\n", average(d1,d2,d3));
   System.out.printf("Average of d1 and d2 and d3 and d4 = %.2f\n",
                     average(d1, d2, d3, d4));
}
```

Variable Length Argument Lists

```
public static double average(double... numbers)
{
    double total = 0.0;

    for (double doozie : numbers)
        total += doozie;

    return total/numbers.length;
}
```

Java treats the variable-length argument list as an array whose elements are all of the same type.

```
public static void PrintIt(char... greeting)
{
    for (char it : greeting)
        System.out.print(it);
    System.out.println();
}

public static void main(String[] args)
{
    char Letters[] = new char [26];

    for (int i = 0; i < Letters.length; i++)
        Letters[i] = (char)(i+65);

    PrintIt(Letters[0]);
}
```

A

```
PrintIt(Letters[7], Character.toLowerCase(Letters[4]),
        Character.toLowerCase(Letters[11]),
        Character.toLowerCase(Letters[11]),
        Character.toLowerCase(Letters[14]));
```

Hello

```
PrintIt(Letters[7], Character.toLowerCase(Letters[4]),
        Character.toLowerCase(Letters[11]),
        Character.toLowerCase(Letters[11]),
        Character.toLowerCase(Letters[14]),
        (char)(Letters[0]-32));
```

Hello!

```
PrintIt(Letters[7], Character.toLowerCase(Letters[8]),
        (char)(Letters[0]-32));
```

Hi!

```
public static void PrintIt(String Prefix, char... greeting)
{
    System.out.print(Prefix);
    for (char it : greeting)
        System.out.print(it);
    System.out.println();
}

public static void main(String[] args)
{
    char Letters[] = new char [26];
    String Prefix = "Greeting ";

    for (int i = 0; i < Letters.length; i++)
        Letters[i] = (char)(i+65);

    PrintIt(Prefix, Letters[7],
      Character.toLowerCase(Letters[8]), (char)(Letters[0]-32));
}
```

Greeting Hi!

Variable Length Argument Lists

```
public static void PrintIt(char... greeting, String Prefix)
```

Placing an ellipsis indicating a variable-length argument list in the middle of a parameter list is a syntax error.

An ellipsis may be placed only at the end of the parameter list.

Object Oriented Programming

Intro to OOP

Going forward…

We are still learning how to program in Java

We will now add studying OO concepts and how to apply them in Java

Other OO languages will have ways of doing almost everything we will do – you are here to learn how to do them in Java

Intro to OOP

- CSE 1310
  - Teaching you how to program
  - Get you in the mindset of a programmer
  - Introduce you to programming
- CSE 1320
  - Learning more advanced programming
  - What goes on behind the scenes (memory, pointers, debugging) using C
- CSE 1325
  - Learning a different type of programming
  - Objects, inheritance, encapsulation using Java

CSE 1310 and 1320 focused on procedural programming.

CSE 1325 will focus on OO programming

OOP

In traditional programming, programs are basically lists of instructions to the computer that define data and then work with that data.

Data and the functions that work on that data are separate entities that are combined together to produce the desired result.

Because of this separation, traditional programming often does not provide a very intuitive representation of reality.

OOP

It’s up to the programmer to manage and connect the properties (variables) to the behaviors (functions) in an appropriate manner. This leads to code that looks like this:

```
driveTo(you, work);
```

A function called `driveTo` that takes `you` and `work` as parameters.

OOP

Object-oriented programming (OOP) provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

This leads to code that looks more like this:

```
you.driveTo(work);
```

You have the ability to drive to work…

OOP

Rather than being focused on writing functions, we focus on defining objects that have a well-defined set of behaviors.

This is why the paradigm is called “object-oriented”.

OOP allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability.

Simple Analogy – The Automobile as an Object

Suppose you want to drive a car and make it go faster by pressing its accelerator pedal.

What must happen before you can do this?

Step 1 - *design* the car

Simple Analogy – The Automobile as an Object

A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. A **class** is like a blueprint in that it is the template for any object created from it.

These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car.

This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Simple Analogy – The Automobile as an Object

Before you can drive a car, it must be *built* from the engineering drawings that describe it.

A completed car has an *actual* gas pedal to make the car go faster.

But the car won't accelerate on its own, so the driver must *press* the gas pedal to accelerate the car.

Simple Analogy – The Automobile as an Object

Gas pedal in our automobile

hides the mechanisms of making the car go faster from the driver

Brake pedal

hides the mechanisms of making the car stop from the driver

Methods

- houses the program statements that actually perform a task
- hides these statements from its user

Methods

`next(), nextLine(), nextInt(), size(), length()`

Simple Analogy – The Automobile as an Object

A car must be built from its drawings/blueprints before you can drive it.

An *object* must be built from a class before it can be used.

This building process is called *instantiation*.

An object is an instance of its class.

A class can be used many times to build many objects (more than one car is built from a drawing)

Simple Analogy – The Automobile as an Object

Pressing on the gas pedal sends a *message* to the car.

Pressing on the brake sends a different *message* to the car resulting in a different action happening.

Calling a method is sending a *message* to an object.

Simple Analogy – The Automobile as an Object

What color is your car? What make is your car? What year is your car?

How many miles are on your car? How much gas is in the tank?

These are all *attributes* of your car. Every car not only has abilities (gas pedal, brake pedal) but each one also has a unique set of *attributes*. My car knows how many miles are on it, but it does not know how many miles are on the car sitting next to it.

An object's attributes are defined in its class's **member variables**.

Car

Attributes?

Abilities?

Pencil Machine

Attributes	**Abilities**
inventoryLevel	buyPencil
changeLevel	displayMoney
pencilCost	showInventoryLevel
	showChangeLevel

Class

In the world of object-oriented programming, we want our types to not only hold data, but provide methods that work with the data as well.

In Java, this is typically done via the **`class`** keyword.

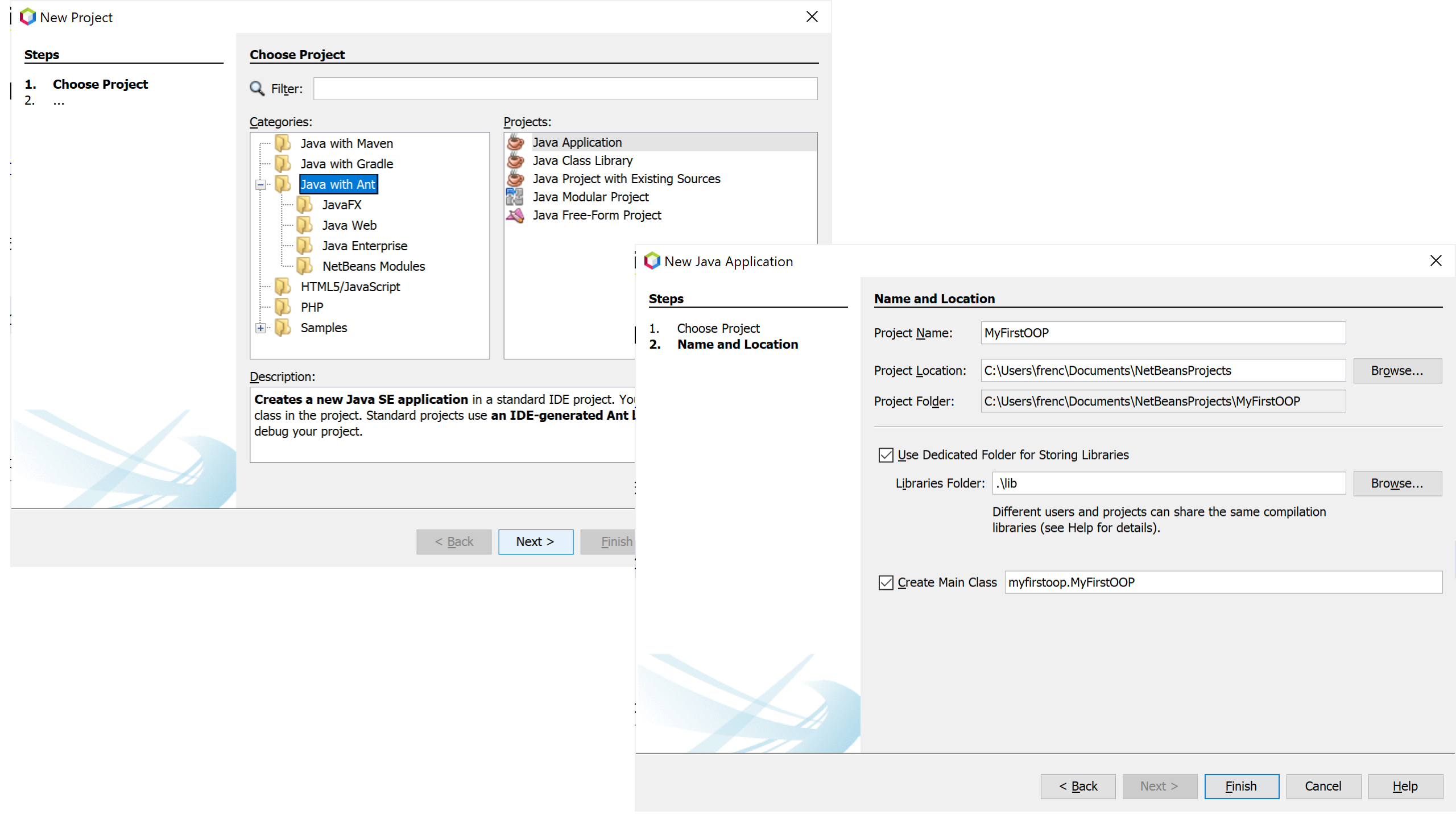
Using the **`class`** keyword defines a new user-defined type called a class.

Struct vs Class

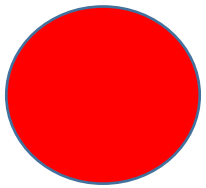
```
struct DateStruct
{
    int year;
    int month;
    int day;
};
```

```
public class DateClass
{
    public int year;
    public int month;
    public int day;
}
```

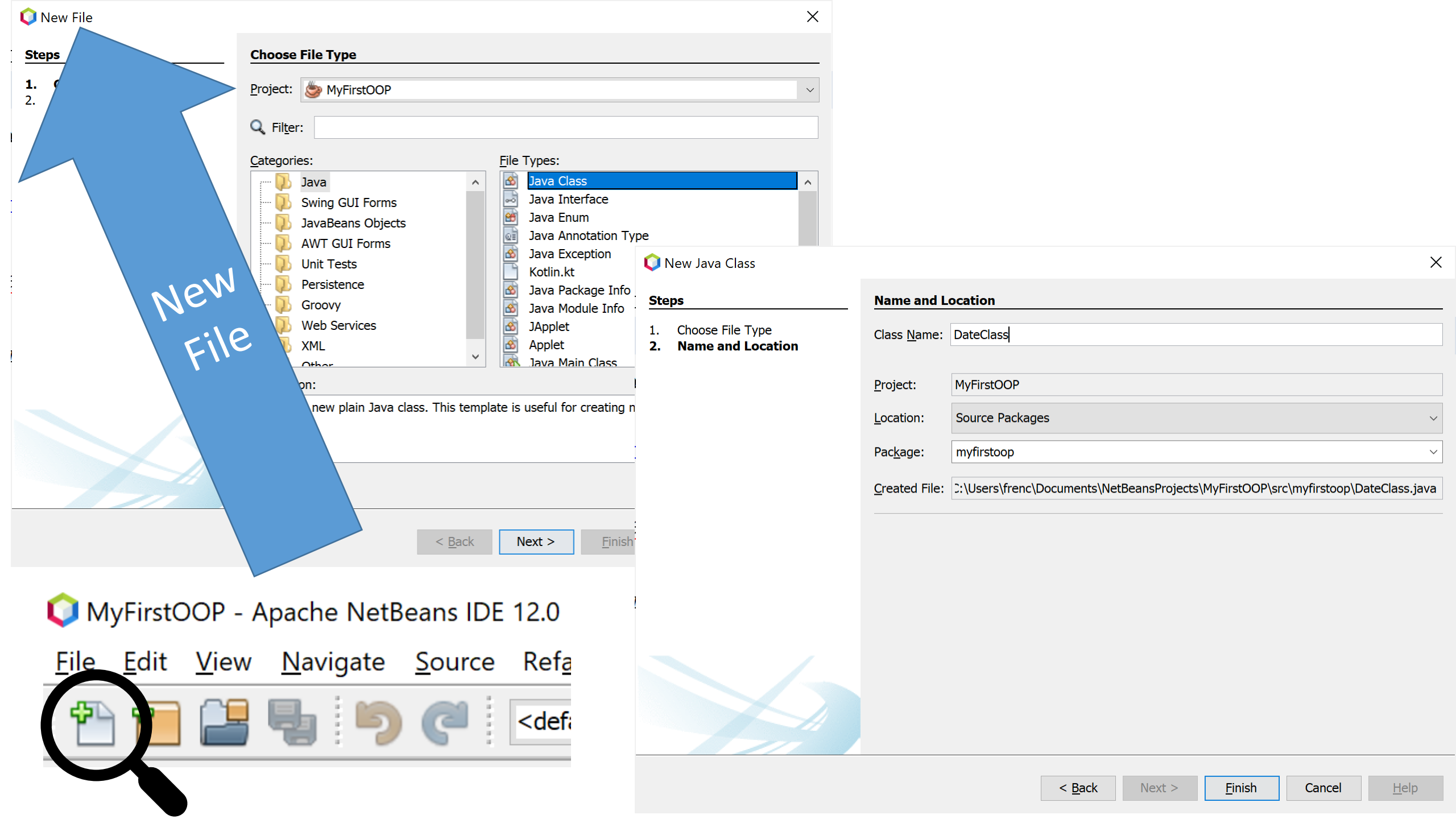
```
public class DateClass
{
    public int year = 2021;
    public int month = 9;
    public int day = 27;

    public void print()
    {
        System.out.printf("%02d/%02d/%4d", month, day, year);
    }
}

DateClass Date = new DateClass();

Date.print();
```

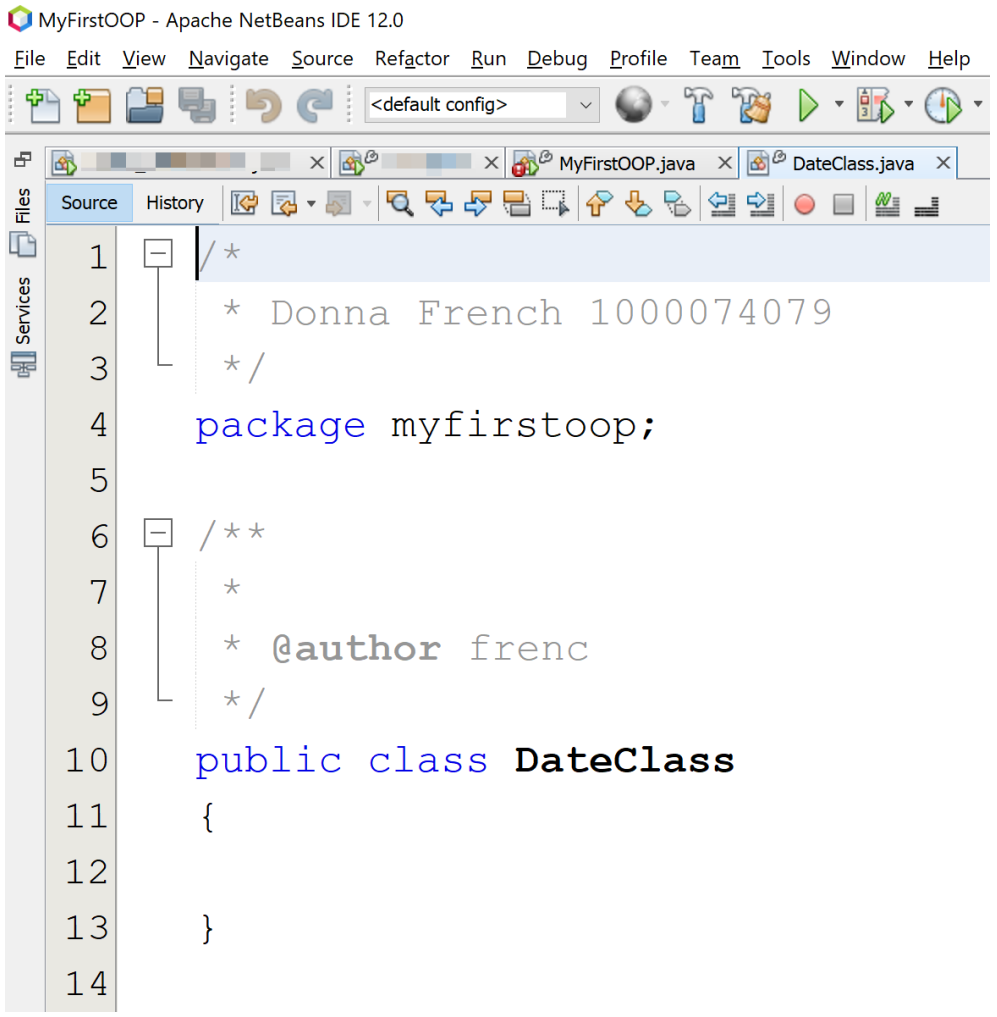

New Project
Steps
1. Choose Project
2. ...
Choose Project
Filter:
Categories:
Java with Maven
Java with Gradle
Java with Ant
JavaFX
Java Web
Java Enterprise
NetBeans Modules
HTML5/JavaScript
PHP
Samples
Projects:
Java Application
Java Class Library
Java Project with Existing Sources
Java Modular Project
Java Free-Form Project
Description:
Creates a new Java SE application in a standard IDE project. Yo
class in the project. Standard projects use an IDE-generated Ant
debug your project.
< Back
Next >
Finish
New Java Application
Steps
1. Choose Project
2. Name and Location
Name and Location
Project Name: MyFirstOOP
Project Location: C:\Users\frenc\Documents\NetBeansProjects
Browse...
Project Folder: C:\Users\frenc\Documents\NetBeansProjects\MyFirstOOP
Use Dedicated Folder for Storing Libraries
Libraries Folder: .\lib
Browse...
Different users and projects can share the same compilation
libraries (see Help for details).
Create Main Class myfirstoop.MyFirstOOP
< Back
Next >
Finish
Cancel
Help

```
package myfirstoop;

public class MyFirstOOP
{
    public static void main(String[] args)
    {
        DateClass Date = new DateClass();

        Date.print();
    }
}
```

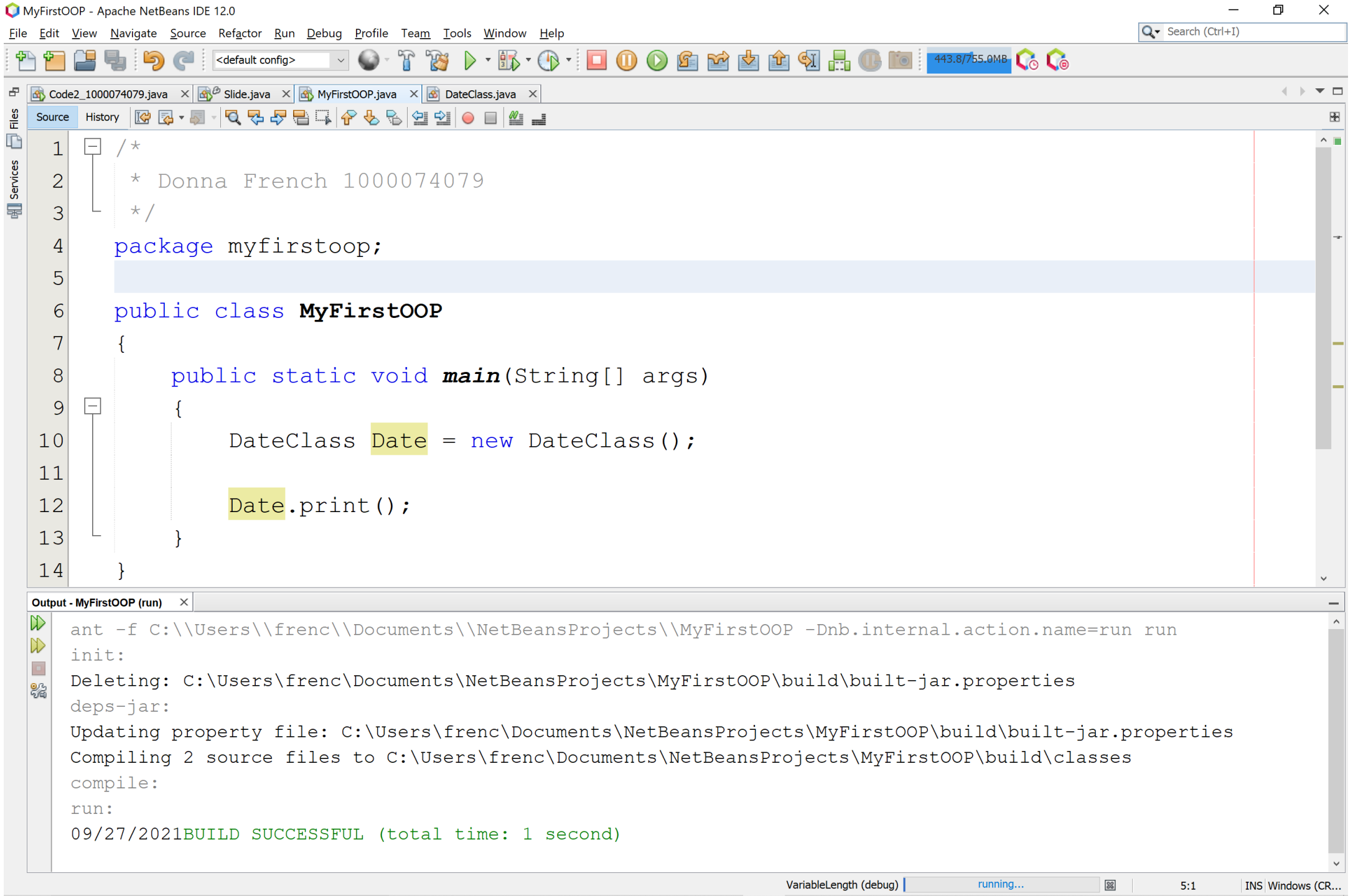


```
cannot find symbol
  symbol:   class DateClass
  location: class MyFirstOOP

cannot find symbol
  symbol:   class DateClass
  location: class MyFirstOOP
----
(Alt-Enter shows hints)
```


New File
Steps
1.
2.
Choose File Type
Project:
MyFirstOOP
Filter:
Categories:
Java
Swing GUI Forms
JavaBeans Objects
AWT GUI Forms
Unit Tests
Persistence
Groovy
Web Services
XML
File Types:
Java Class
Java Interface
Java Enum
Java Annotation Type
Java Exception
Kotlin.kt
Java Package Info
Java Module Info
JApplet
Applet
new plain Java class. This template is useful for creating n
< Back
Next >
Finish
New Java Class
Steps
1. Choose File Type
2. Name and Location
Name and Location
Class Name:
DateClass
Project:
MyFirstOOP
Location:
Source Packages
Package:
myfirstoop
Created File:
:\Users\frenc\Documents\NetBeansProjects\MyFirstOOP\src\myfirstoop\DateClass.java
< Back
Next >
Finish
Cancel
Help
MyFirstOOP - Apache NetBeans IDE 12.0
File Edit View Navigate Source Refa




```
/*
 * Donna French 1000074079
 */
package myfirstoop;

public class DateClass
{
    public int year = 2021;
    public int month = 9;
    public int day = 27;

    public void print()
    {
        System.out.printf("%02d/%02d/%4d",
                          month, day, year);
    }
}
```


MyFirstOOP - Apache NetBeans IDE 12.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
443.8/755.0MB
Code2_1000074079.java
Slide.java
MyFirstOOP.java
DateClass.java
Source History
1 /*
2 * Donna French 1000074079
3 */
4 package myfirstoop;
5
6 public class MyFirstOOP
7 {
8 public static void main(String[] args)
9 {
10 DateClass Date = new DateClass();
11
12 Date.print();
13 }
14 }
Output - MyFirstOOP (run)
ant -f C:\\Users\\frenc\\Documents\\NetBeansProjects\\MyFirstOOP -Dnb.internal.action.name=run run
init:
Deleting: C:\Users\frenc\Documents\NetBeansProjects\MyFirstOOP\build\built-jar.properties
deps-jar:
Updating property file: C:\Users\frenc\Documents\NetBeansProjects\MyFirstOOP\build\built-jar.properties
Compiling 2 source files to C:\Users\frenc\Documents\NetBeansProjects\MyFirstOOP\build\classes
compile:
run:
09/27/2021BUILD SUCCESSFUL (total time: 1 second)
VariableLength (debug)
running...
5:1
INS Windows (CR...

MyFirstOOP.java

```
package myfirstoop;

public class MyFirstOOP
{
    public static void main(String[] args)
    {
        DateClass Date = new DateClass();

        Date.print();
    }
}
```

DateClass.java

```
package myfirstoop;

public class DateClass
{
    public int year = 2021;
    public int month = 9;
    public int day = 27;

    public void print()
    {
        System.out.printf("%02d/%02d/%4d", month, day, year);
    }
}
```

Class

The associated object is essentially implicitly passed to the method.

For this reason, it is often called the implicit object.

The key point is that with non-class methods, we have to pass data to the method to work with.

With method, we can assume we always have an implicit object of the class to work with.

Class

So when you instantiate a class

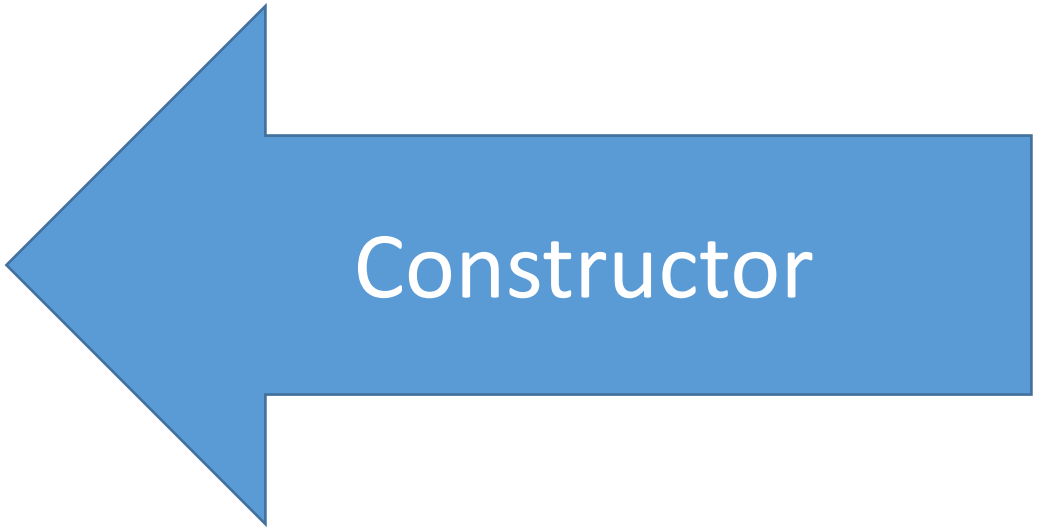
```
DateClass Date = new DateClass();
```

and then you call a member function of that class

```
Date.print();
```

It is assumed that you want to print the data inside that object

```
public class DateClass
{
    public int year = 2021;
    public int month = 9;
    public int day = 27;

    DateClass(int m, int d, int y)
    {
        year = y;
        month = m;
        day = d;
    }

    public void print()
    {
        System.out.printf("%02d/%02d/%4d", month, day, year);
    }
}
```



```
package myfirstoop;

public class MyFirstOOP
{
    public static void main(String[] args)
    {
        DateClass Date1 = new DateClass(9,27,2021);
        DateClass Date2 = new DateClass(1,1,2021);

        Date1.print();
        System.out.println();
        Date2.print();
        System.out.println();
    }
}
```

```
09/27/2021
01/01/2021
```

Intro to OOP

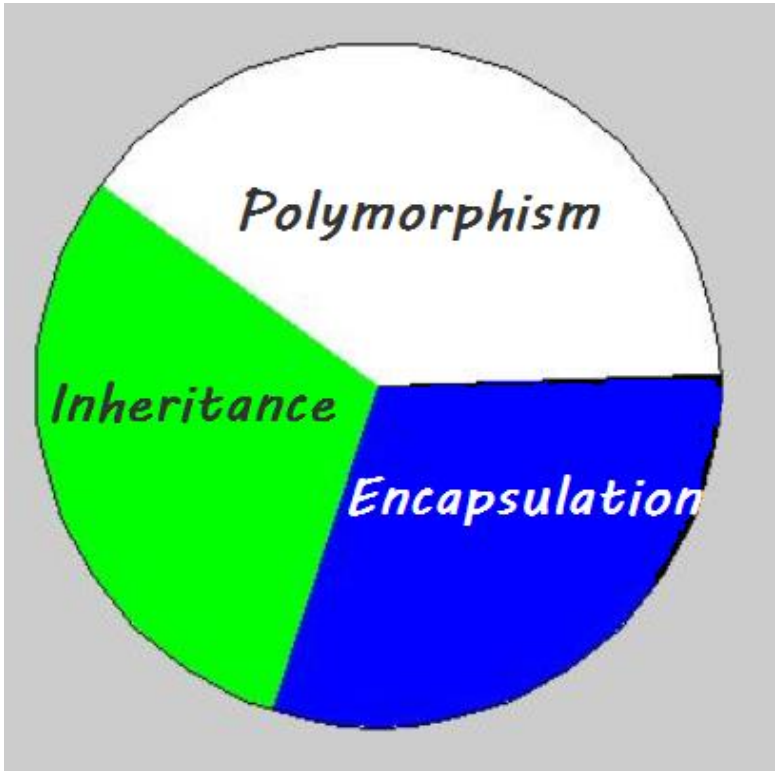
Three main concepts of Object Oriented Programming

OO PIE

Polymorphism

Inheritance

Encapsulation



OOP Vocabulary

Polymorphism

describes the ability of different objects to be accessed by a common interface

Inheritance

describes the ability of objects to derive behavior patterns from other objects

These objects can then customize and add new unique characteristics

Encapsulation

describes the ability of objects to maintain their internal workings independently from the program they are used in

Classes **encapsulate** attributes and methods into objects created from those classes

Classes

- Each class you create becomes a new type that can be used to declare variables and create objects.
- Objects know things and how to do things
- You can declare new classes as needed; this is one reason Java is known as an extensible language.

Classes

Let's create a new class named Account (think Bank Account)

What are some of the things an Account would know and be able to do?

attributes

- name
- account number
- balance

abilities

- set/update the name
- retrieve the name
- set/update the account number
- retrieve the account number
- set/update the balance
- retrieve the balance

Classes

Those attributes are called **instance** variables.

A class is just a definition of a type; therefore, does not use any memory until you create (instantiate) an object of that class.

Variables are defined in the class, but have no memory allocated to them either

Once you instantiate an object, those variables now have memory

That's why they are called **instance** variables

Classes

Let's look at our Account's abilities

- set/update the name
- retrieve the name
- set/update the account number
- retrieve the account number
- set/update the balance
- retrieve the balance

Notice a pattern here?

For each attribute, we have an ability to

set/update it
retrieve it

In OOP terms, these abilities are generically called

getters

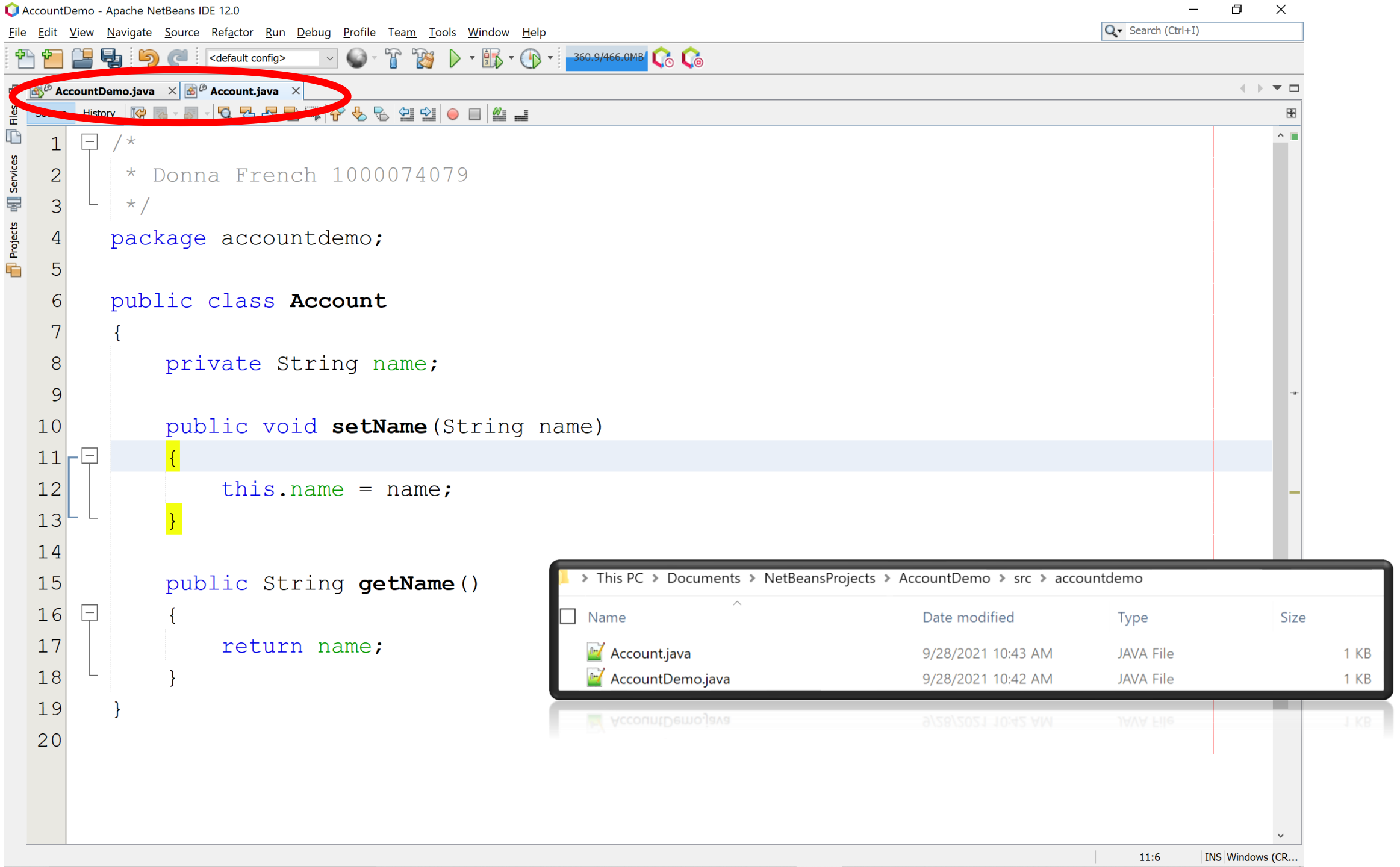
setters

```
package accountdemo;
```

AccountDemo.java

```
public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Each class declaration that begins with the access modifier `public` must be stored in a file that has the same name as the class and ends with the `.java` filename extension.


AccountDemo - Apache NetBeans IDE 12.0
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
360.9/466.0MB
AccountDemo.java
Account.java
History
Files
Services
Projects
1 /*
2 * Donna French 1000074079
3 */
4 package accountdemo;
5
6 public class Account
7 {
8 private String name;
9
10 public void setName(String name)
11 {
12 this.name = name;
13 }
14
15 public String getName()
16 {
17 return name;
18 }
19 }
20
This PC › Documents › NetBeansProjects › AccountDemo › src › accountdemo
Name
Date modified
Type
Size
Account.java
9/28/2021 10:43 AM
JAVA File
1 KB
AccountDemo.java
9/28/2021 10:42 AM
JAVA File
1 KB
11:6
INS
Windows (CR...

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Every class declaration contains keyword `class` followed immediately by the class's name.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

An object has attributes that are implemented as instance variables and carried with it throughout its lifetime.

Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Each object (instance) of the class has its own copy of each of the class's instance variables.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

It is preferrable to list the class's instance variables first inside the class.

This is not required but scattering them throughout the class can lead to hard to follow classes.

Instance variables MUST be listed outside of the class's methods.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Access Modifiers

`public` and `private`

Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they are declared.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Instance Method

`setName`

Up to now, you have declared `static` methods.

A class's non-`static` methods are known as instance methods.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Instance method `setName`'s declaration indicates that `setName` receives parameter `name` of type `String`—which represents the name that will be passed to the method as an argument.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Variables declared in the body of a particular method are local variables and can be used only in that method and that a method's parameters also are local variables of the method.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Method `setName`'s body contains a single statement that assigns the value of the name *parameter* (a `String`) to the class's `name` *instance variable*, thus storing the account name in the object.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

If a method contains a local variable with the same name as an instance variable, that method's body will refer to the local variable rather than the instance variable.

In this class, the local variable is said to *shadow* the instance variable in the method's body.

The method's body can use the keyword `this` to refer to the shadowed instance variable explicitly.

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

We could have avoided using `this` if we had chosen a different variable name for the parameter.

It is a widely accepted practice in OOP to reuse the same name in order to minimize the number of variables used.

Do not add `this` to all uses of your class members.

Only do so when you have a specific reason to.

We will see more examples of when using `this` is necessary.

getName

```
package accountdemo;

public class Account
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

Instance method `getName` returns a particular Account object's name to the caller.

The method has an empty parameter list because it does not require additional information to perform its task.

In Class Exercise

Open a file named "Skittles.txt" and print the first line of the file to the screen.

Action Items

Wed, Sep 28

Due 11:59pm OLQ5

Mon, Oct 3

Due 11:59pm Crash Course : Quiz 5

Due 11:59pm Coding Assignment 2

Wed, Oct 5

Due 11:59pm Homework 3