AP CS A (Java) Training

Unit 3: Class Creation

Updated Curriculum (2025-2026)

3.1 Abstraction and Program Design

Learning Objectives:

- Represent the design of a program by using natural language or creating diagram that indicates the classes in the program and the data and procedural abstractions found in each class by including all attributes and behaviors.

Key Terms to Know

- **Abstraction**: Is the process of reducing complexity of systems by focusing on their main idea.
- This is done by hiding unnecessary details, which makes it easier to manage them. Some types of abstractions include:
    - Data abstraction(**Ex. Color and character representation of binary**)
    - Hardware abstraction(**Turn off your TV by pressing power button from your wireless remote**)
    - Software abstraction (**Calling a method by simply referring to its name**)
- An **attribute** is a type of data abstraction that is defined in a class outside of any method or constructor.
- An **instance variable** is is an attribute whose value is unique to each instance of the class.
- A **class variable** is an attribute shared by all instances of the class.
    - Recall: A class variable is indicated by the static keyword during its declaration.

Key Terms to Know

- **Procedure abstraction** is the process of calling a method by simply referring to its name without having to know the details on how such a method was implemented. You simply need to know what it does rather than how it does it.
- **Method decomposition** allows programmers to break down larger behaviors into smaller behaviors by creating methods to represent each individual smaller behavior.
- This allows for code reuse which helps manage complexity.
- **Parameters** allow procedures to be generalized by enabling the the procedures to be reused with a range of input values or arguments.
- Before implementing a class, it is important to **take time to design** each class including its attributes and behaviors which can be represented using natural language or diagram.

Abstraction

Abstraction is the process of hiding unnecessary details and showing only the essential features of something.

- When you use a phone, you press buttons to call. You don't need to know how the electronics inside work.
- Similarly in programming, you can use a method or a class without knowing exactly how it works inside.

➢ **Purpose of Abstraction:**

- Simplifies complex systems.
- Focuses on *what* an object does, not *how* it does it.

➢ **Example in Daily Life:**

- **A TV remote:** You know which button changes the channel, but not how the signal is processed.
- **Driving a car:** When you drive a car, you only use the steering wheel, pedals, and gear. You don't need to understand how the engine, brakes, or transmission work internally.

Program Design

Program design is the process of planning how a program will be built. It involves thinking about the structure, the components (like classes and methods), and how they work together.

Steps in Program Design:

1. **Understand the Problem:** What should the program do?
2. **Identify Components:** Break it into smaller parts (objects/classes).
3. **Apply Abstraction:** Hide unnecessary internal details.
4. **Design Interactions:** How components talk to each other.
5. **Plan Data:** What data is needed and where it will be stored.
6. **Test and Improve:** Make sure it works and improve it if needed.

How Abstraction Connects to Program Design

- Abstraction is a **key tool in program design**.
- It helps programmers focus on the main tasks without getting overwhelmed by small internal details.
- Makes programs easier to read, maintain, and expand.

ATM Machine Design

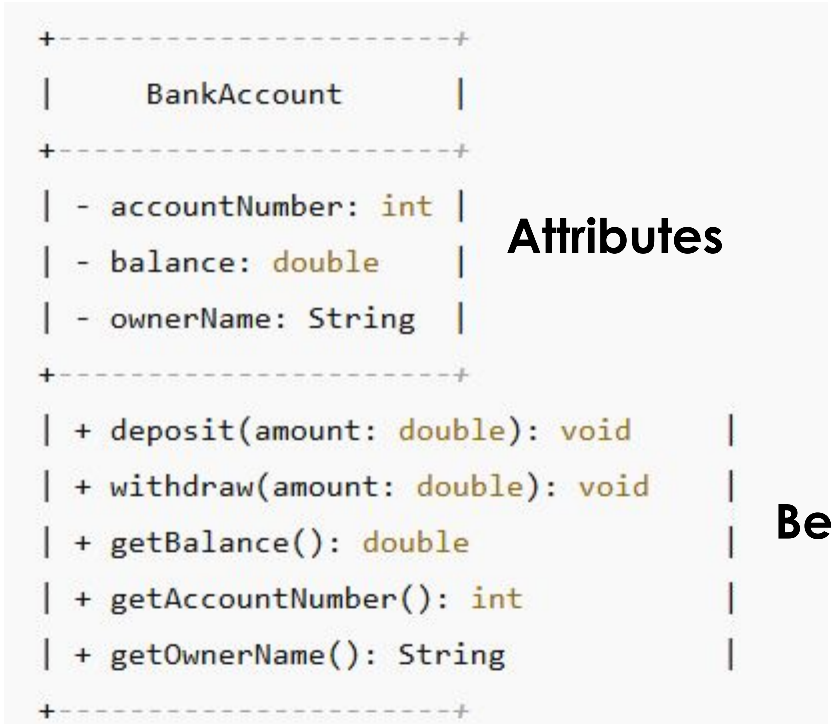
Abstraction:

- Users only see buttons like **Withdraw**, **Deposit**, **Check Balance**.
- Users don't see how data is fetched, verified, or how transactions are processed internally.

Program Design Steps:

- **Classes:** `ATM`, `BankAccount`, `Transaction`
- **Methods:** `withdraw()`, `deposit()`, `checkBalance()`
- Design how these classes interact to fulfill user actions.

Example of a BankAccount Class Design using UML diagram



```
+----------------------+
|     BankAccount      |
+----------------------+
| - accountNumber: int |
| - balance: double    |
| - ownerName: String  |
+----------------------+
| + deposit(amount: double): void     |
| + withdraw(amount: double): void    |
| + getBalance(): double              |
| + getAccountNumber(): int           |
| + getOwnerName(): String            |
+----------------------+
```

Attributes

Behaviors

3.2 Impact of Program Design

Learning Objectives:

- Explain the social and ethical implications of computing systems

Key Terms to Know

- **System reliability** refer to the program being able to perform its tasks as expected under stated conditions without failure.
- Programmers should make an effort to maximize system reliability by testing the program with a variety of conditions.
- The creation of programs has impacts on society, the economy, and culture.
- These impacts can be both positive and negative.
- Programs meant to fill a need or solve a problem can have unintended consequences beyond their intended use.
- Legal issue and intellectual property concerns also arises when creating program.
- Programmers often reuse code written by others and published as open source and free to use.
- Incorporation of code that is not published as open source requires programmers to obtain permission and often purchase the code before integrating it into it into program.

Social and Ethical Implications of Computing Systems

Social Implications

These are the ways computing systems affect society, communities, and people's lives.

Positive Impacts:

- **Improved Communication:** Social media, messaging apps, and video calls connect people worldwide.
- **Access to Information:** The internet makes education, news, and knowledge widely accessible.
- **Healthcare Advances:** Medical technologies, online consultations, and health monitoring.
- **Remote Work and Learning:** Enables people to work or study from anywhere.

Social and Ethical Implications of Computing Systems

Negative Impacts:

- **Job Displacement:** Automation and AI can replace some jobs.
- **Digital Divide:** Not everyone has equal access to technology or the internet.
- **Cyberbullying and Harassment:** Social media misuse can harm individuals.
- **Addiction and Mental Health Issues:** Overuse of devices and apps can negatively affect well-being.
- **Privacy Loss:** Personal data is often collected without full consent.

Ethical Implications of Computing Systems

Privacy:

- Is it ethical for companies to track users without clear consent?
- Example: Websites collecting user data without permission.

Security:

- Who is responsible when a security breach happens?
- Protecting personal and financial data is a major responsibility.

Bias and Fairness:

- Algorithms can be biased if they are trained on unfair data.
- Example: Using AI in hiring, lending, etc.

Ethical Implications of Computing Systems

Intellectual Property:

- Is it right to download or use software, music, or videos without paying?
- Respecting copyrights and creators' rights.

Digital Responsibility:

- How should users behave online?
- Includes avoiding plagiarism, cyberbullying, and spreading unverified information.

Environmental Impact:

- Computers and data centers use lots of energy and create electronic waste.
- Ethical responsibility includes designing eco-friendly systems.

In summary:

- We are living in a world where technology innovations are happening at an unprecedented pace.
- With those fascinating innovations, there are consequences (beneficial and harmful) that will tremendously impact the society.
- Ethical and social implications of computing systems should be of great concerns
- Computing system is not perfect, so their reliability should be taken seriously by programmers in order to continue enhancing them.
- When creating software applications, legal and intellectual implications can also arise.
- Now, with the rapid integration of AI and machine learning, there are more concerns as how the future of the world will be transformed.
- Overall, the creation of software programs will continue to have great impact on the society, economies, and culture. With that in mind, programmers should take these issues into consideration when creating software applications.

3.3 Anatomy of a Class

Learning Objectives:

- Develop code to designate access and visibility constraints to classes, data, constructors, and methods.
-

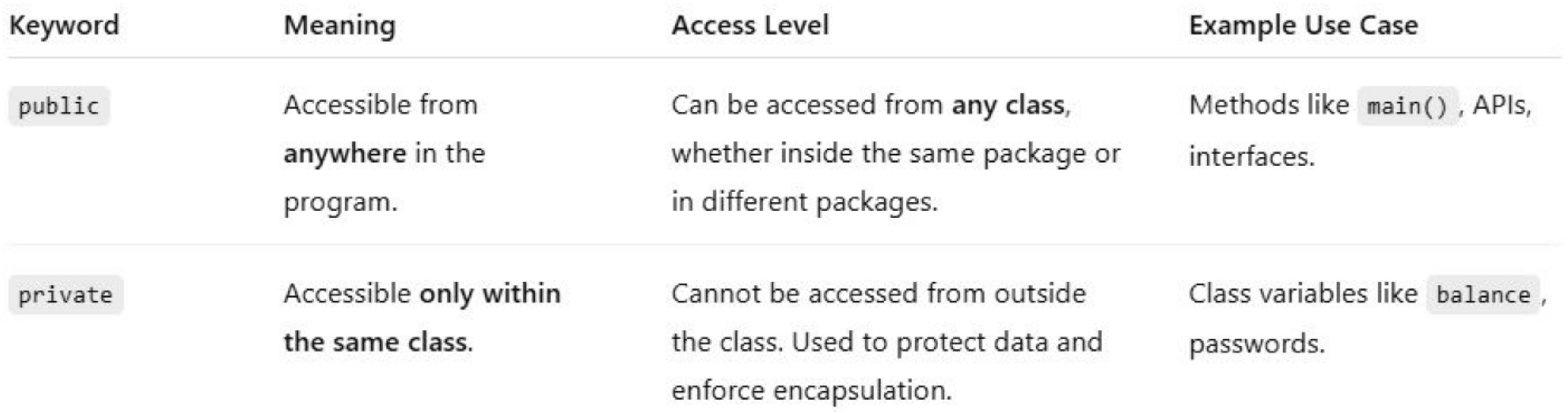
Key Terms to Know

- **Data encapsulation:**
  - Technique in which the implementation of details of a class are kept hidden from external classes.
  - The keywords **public** and **private** affect the access of classes, data, constructors, and methods.
  - In this course, classes are always designated as public and are associated with the keyword class.
- **Public**
  - Allows access from classes outside the declaring class.
- **Private**
  - Restricts access to the declaring class

Key Terms to Know

- **Instance variables**
    - Belong to the object, and each object has its own copy of the variable
    - Access to attributes should be kept internal to the class in order to accomplish encapsulation.
    - For this reason, it is a good programming practice to designate instance variables for these attributes as private unless otherwise specified.
    -

Public vs. private access modifiers



Keyword	Meaning	Access Level	Example Use Case
`public`	Accessible from **anywhere** in the program.	Can be accessed from **any class**, whether inside the same package or in different packages.	Methods like `main()`, APIs, interfaces.
`private`	Accessible **only within the same class**.	Cannot be accessed from outside the class. Used to protect data and enforce encapsulation.	Class variables like `balance`, passwords.

Anatomy of a Class

- A **class** is a blueprint from which objects are created in Java.
- The first letter of a class name is always capitalized.
- The **public** and **private** keywords determine the visibility of classes, data, constructors, and methods.
- In object-oriented programming design, data of a class should be accessible only within that class. For this reason, instance variables are declared as private.
- Constructors are declared as public
- Classes are declared as public
- Methods, which often define the behaviors of an object of a class, are declared as either public or private.

Public Access

Rectangle.java

```
public class Rectangle
{
     /*public methods, variables and
     constructors written in
     Rectangle can be used in
     MyProgram.java and in
     Rectangle.java*/


}
```

MyProgram.java

Private Access

Rectangle.java

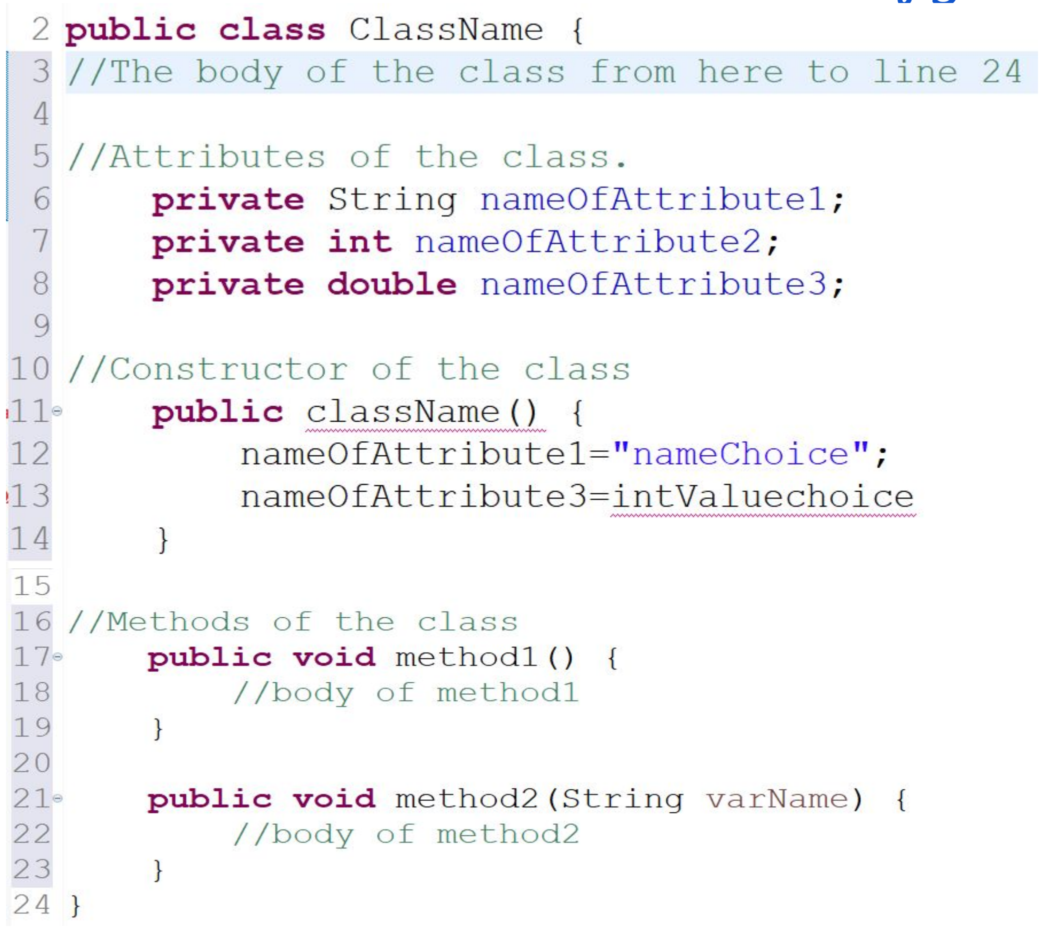
```
public class Rectangle
{
     /*private methods, variables and
     constructors written in
     Rectangle cannot be used in
     MyProgram.java, but can be used
     and accessed in Rectangle.java*/


}
```

MyProgram.java

Why private access???

- So why would we want to make any of our data private? We use private access for methods and data in order to manage complexity by controlling how users can interact with objects and hiding implementation details that are unnecessary for users.
- This process of hiding the implementation details of a program is referred to as **encapsulation**. This is one of the core principles of Object Oriented Programming.
- **Encapsulation** is different than Abstraction because it is concerned with hiding the internal state of an object, such as its instance variables, whereas **abstraction** is concerned with hiding the details of how something works.

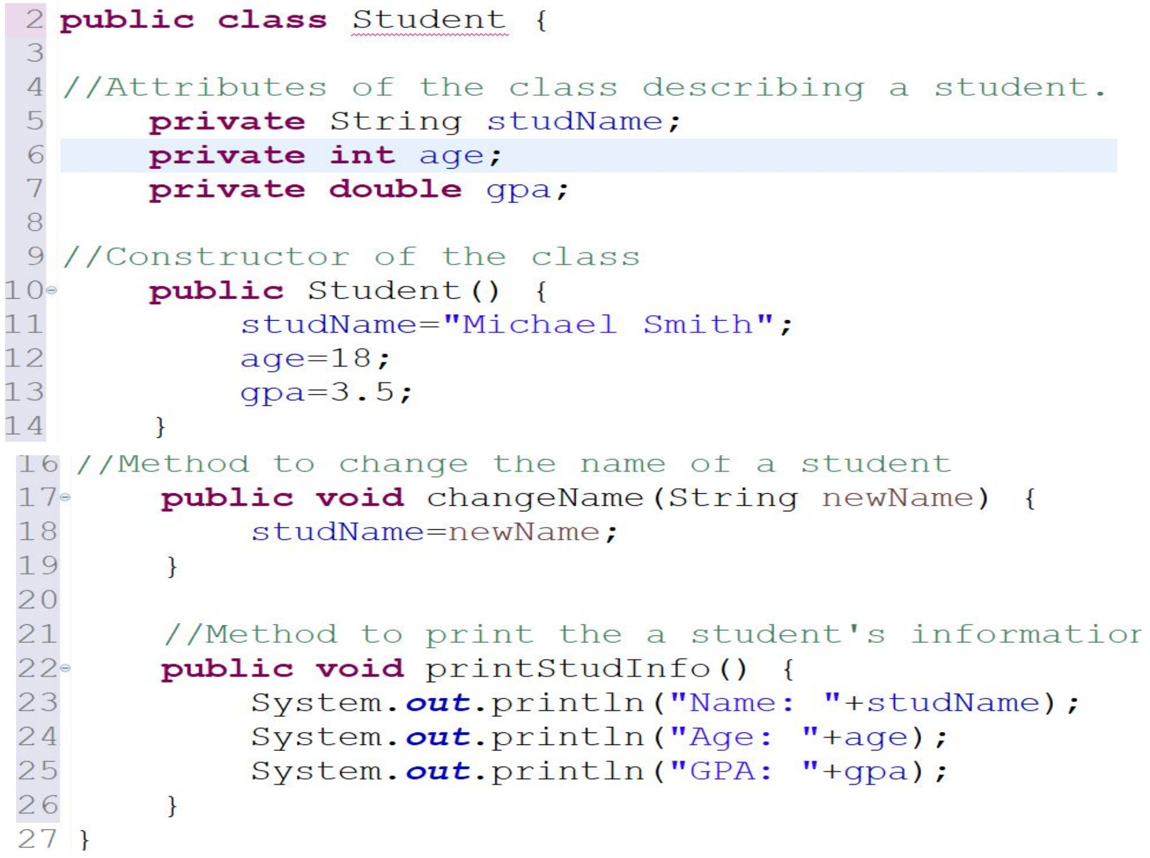
Anatomy of a Class



```
 2 public class ClassName {
 3 //The body of the class from here to line 24
 4 
 5 //Attributes of the class.
 6     private String nameOfAttribute1;
 7     private int nameOfAttribute2;
 8     private double nameOfAttribute3;
 9 
10 //Constructor of the class
11     public className() {
12         nameOfAttribute1="nameChoice";
13         nameOfAttribute3=intValuechoice
14     }
15 
16 //Methods of the class
17     public void method1() {
18         //body of method1
19     }
20 
21     public void method2(String varName) {
22         //body of method2
23     }
24 }
```

Anatomy of a Class (Example)

Below is an example of a Student class with its attributes, constructor, and behaviors



```java
public class Student {

//Attributes of the class describing a student.
    private String studName;
    private int age;
    private double gpa;

//Constructor of the class
    public Student() {
        studName="Michael Smith";
        age=18;
        gpa=3.5;
    }
//Method to change the name of a student
    public void changeName(String newName) {
        studName=newName;
    }

    //Method to print the a student's informatior
    public void printStudInfo() {
        System.out.println("Name: "+studName);
        System.out.println("Age: "+age);
        System.out.println("GPA: "+gpa);
    }
}
```

3.4 Constructors

Learning Objectives:

- Develop code to declare instance variables for the attributes to be initialized in the body of the constructors of a class.

Key Terms to Know

- **Object's state**
  - Refers to its attributes and their values at a given time and is defined by instance variables belonging to the object.
  - This establishes a has-a relationship between the object and its instance variables.
- **Constructor**
  - Must have the same name as the class
  - Must not have any return type(Not even **void**)
  - Is used to set the initial state of an object, which should include initial values for all instance variables.
  - When a constructor is called, memory is allocated for the object.
  - When no constructor is defined, Java provides a **non-parameter** constructor, and the instance variables are set to the default values according to the data type of the attribute. This constructor is referred to as the default constructor.
- **Default constructor:**
  - Is a constructor with no parameter.

Writing Constructors

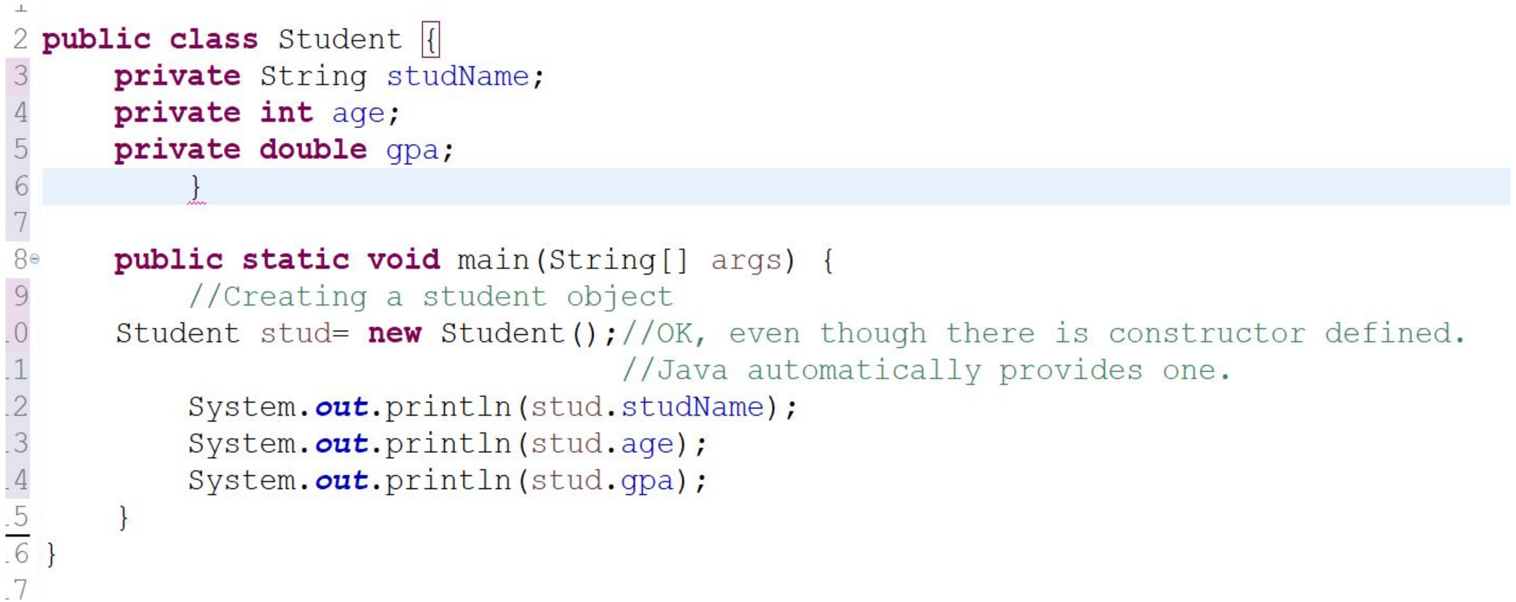
- When writing a class, constructors usually start after the instance variables but before the methods (think about the `Student` class we made in the previous slides)
- They typically start with a public and must contain the class's name as the "name" of the constructor
  - `public Student()`
- Parameters can go inside the parentheses to allow us to set instance variables
  - `public Student(String studName, int age, double gpa)`
- Classes usually have at least two constructors - one that takes in no parameters, and another that takes all the parameters necessary to set all the instance variables of the object
  - let's look at an example...

Constructors(default values of instance variables)

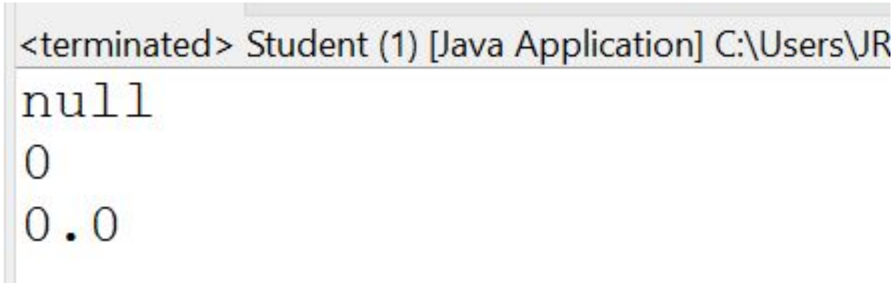
IMPORTANT TO REMEMBER:

- A default constructor is a constructor with no parameter
- When no constructor is written by the programmer, Java provides a no-argument default constructor, and the instance variables are set to the java default values for that datatype (remember this ***ONLY*** happens if no constructor is written at all, if there is/are constructor(s) written, then those are the only ones available) :
    - **`0`** for **`int;`** **`0.0`** for **`double`**; **`false`** for **`boolean;`** **`null`** for any object datatype like **`String`**

Constructors (Default) (Currently Not defined by the programmer)



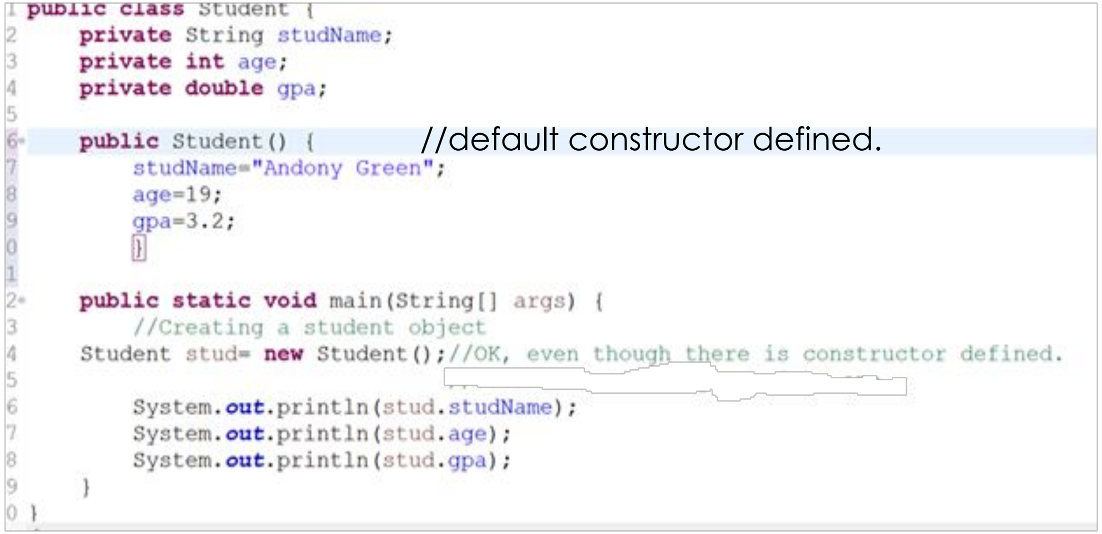
```java
public class Student {
    private String studName;
    private int age;
    private double gpa;
        }

    public static void main(String[] args) {
        //Creating a student object
    Student stud= new Student();//OK, even though there is constructor defined.
                                //Java automatically provides one.
        System.out.println(stud.studName);
        System.out.println(stud.age);
        System.out.println(stud.gpa);
    }
}
```



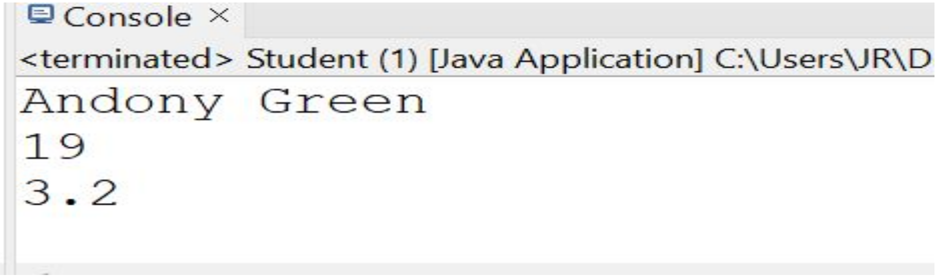
```
<terminated> Student (1) [Java Application] C:\Users\JR
null
0
0.0
```

Default Constructor manually defined

- Programmers can manually create the default constructor

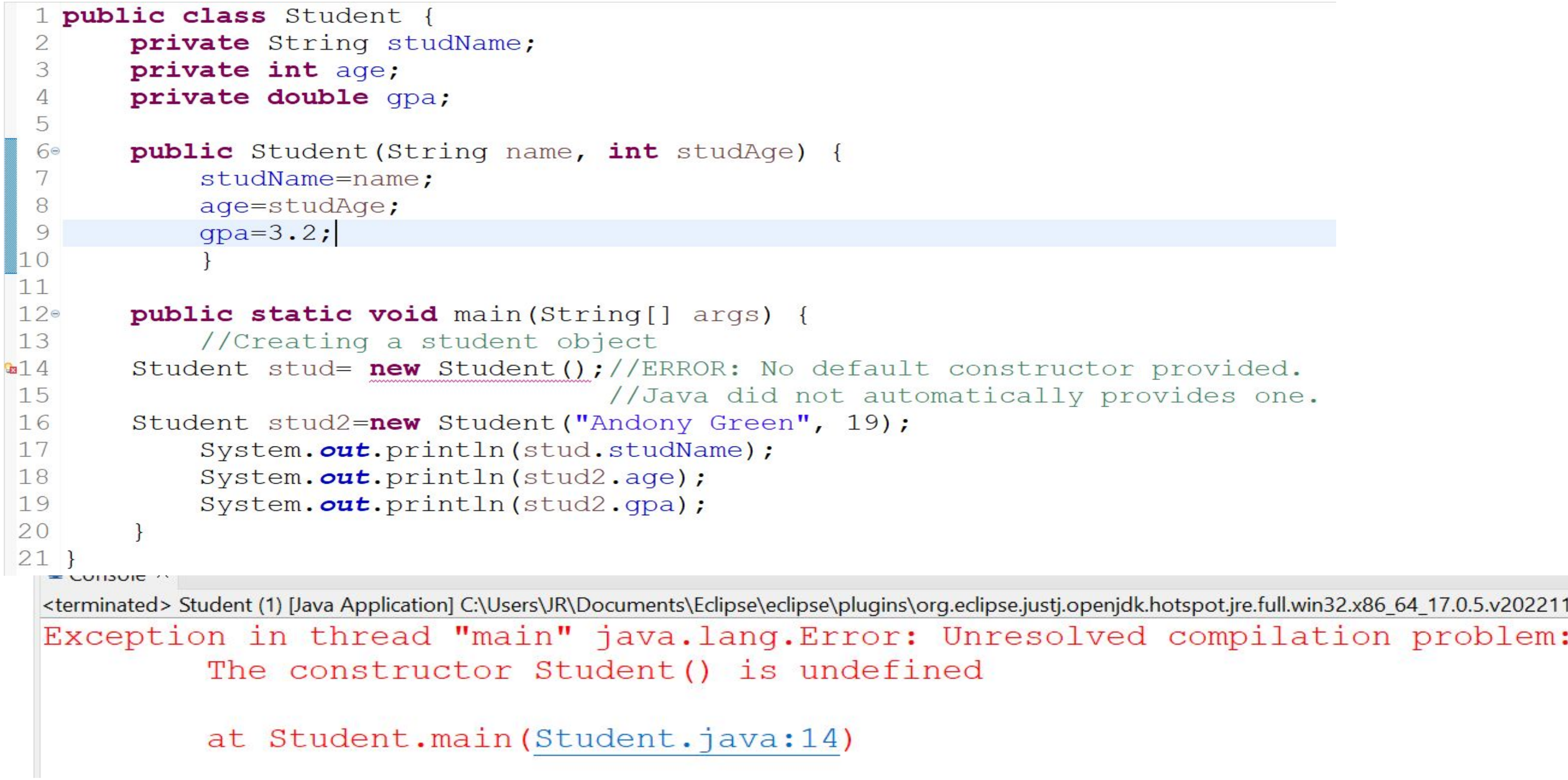


```
public class Student {
    private String studName;
    private int age;
    private double gpa;

    public Student() {          //default constructor defined.
        studName="Andony Green";
        age=19;
        gpa=3.2;
        }

    public static void main(String[] args) {
        //Creating a student object
    Student stud= new Student();//OK, even though there is constructor defined.

        System.out.println(stud.studName);
        System.out.println(stud.age);
        System.out.println(stud.gpa);
    }
}
```



```
Console ×
<terminated> Student (1) [Java Application] C:\Users\JR\D
Andony Green
19
3.2
```

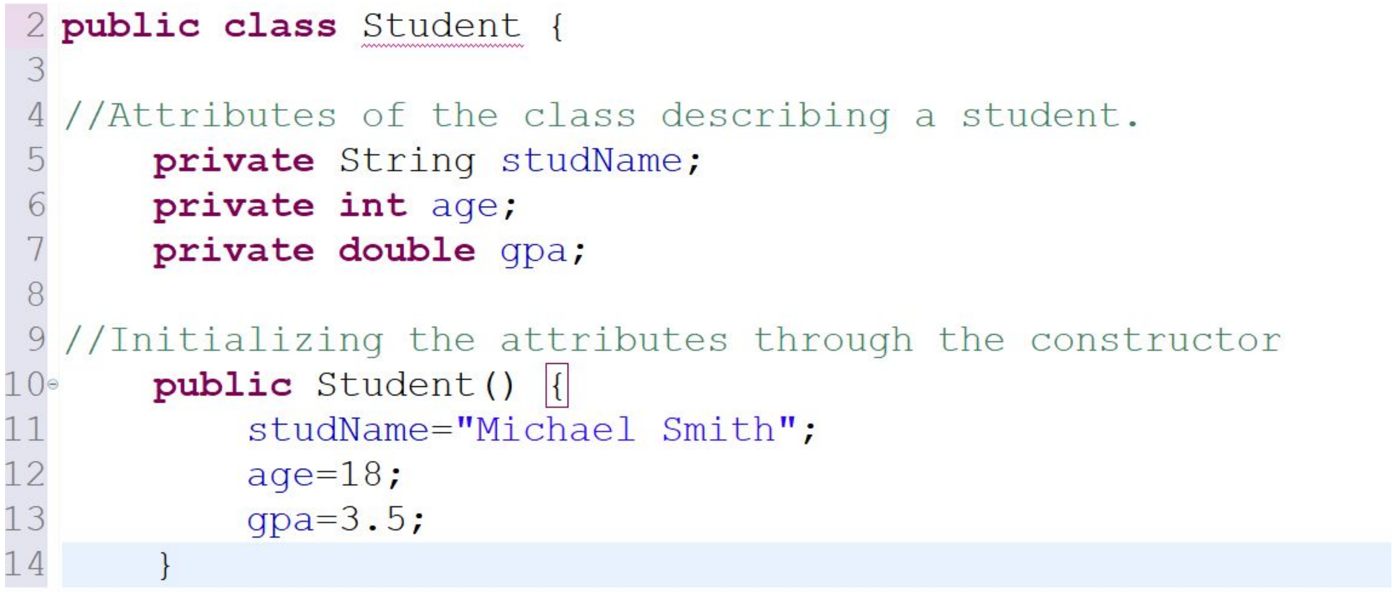
Default Constructor Access Error



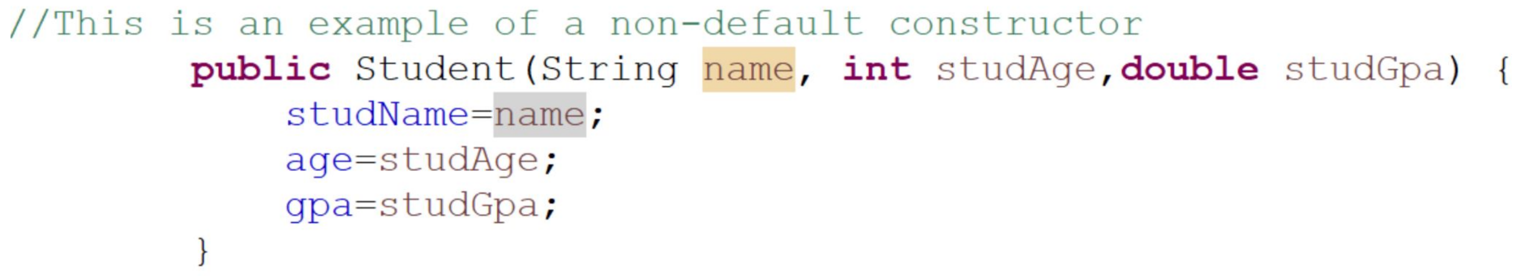
```java
public class Student {
     private String studName;
     private int age;
     private double gpa;

     public Student(String name, int studAge) {
          studName=name;
          age=studAge;
          gpa=3.2;
          }

     public static void main(String[] args) {
          //Creating a student object
     Student stud= new Student();//ERROR: No default constructor provided.
                                 //Java did not automatically provides one.
     Student stud2=new Student("Andony Green", 19);
          System.out.println(stud.studName);
          System.out.println(stud2.age);
          System.out.println(stud2.gpa);
     }
}
```

```
<terminated> Student (1) [Java Application] C:\Users\JR\Documents\Eclipse\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.5.v20221
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The constructor Student() is undefined

	at Student.main(Student.java:14)
```

OVERLOADED CONSTRUCTORS(2 OR MORE CONSTRUCTORS DEFINED)



```java
public class Student {

//Attributes of the class describing a student.
    private String studName;
    private int age;
    private double gpa;

//Initializing the attributes through the constructor
    public Student() {
        studName="Michael Smith";
        age=18;
        gpa=3.5;
    }
```



```java
//This is an example of a non-default constructor
        public Student(String name, int studAge,double studGpa) {
            studName=name;
            age=studAge;
            gpa=studGpa;
        }
```

3.5 Methods: How to Write Them

Learning Objectives:

- Develop code to define behaviors of an object through methods written in a class using primitive values and determine the result of calling these methods

Key Terms to Know

- **Void method**
  - Does not return a value, and its header contains the keyword void before its name.
- **Non-void method:**
  - Return a single value, and its header includes the return type instead of the keyword void.
  - In a non-void method, the return type indicated in the header must match the type of the value being returned.
  - The return keyword is used to return the flow of control to the point where the method or constructor was called, and any code that is sequentially after a return statement will never be executed.
    - Executing a return statement inside a selection or iteration statement will halt the statement and exit the method or constructor
- **Accessor method:** Allows objects of other classes to obtain a copy of the value of instance variables or class variables. An accessor method is a non-void method.
- **Mutator method**: is a method that changes the value of the instance or class variables.
  - It is often defined as a void method.

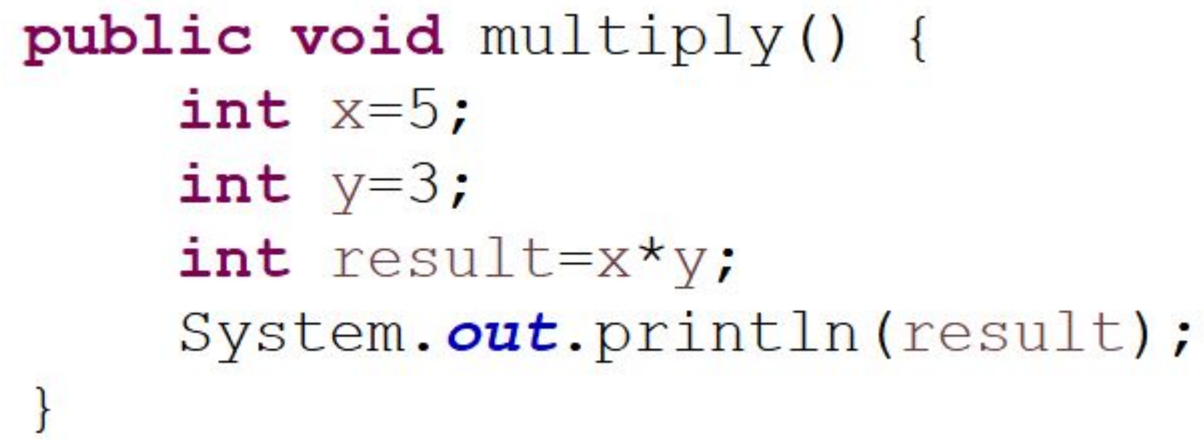
Writing Methods

- A method is a block of code with a name that can be called to execute anywhere in the program.
- Methods promote code reusability and help make programs easier to debug.
- A method is made of **access modifier**(optional), **return type**, **method name**, **parameter**(s), **method body**, and **return value**.
- Below is a structure of a method in Java.

```
AccessModifier returnType methodName(Param List(optional)) {

    //Body of the method

}
```

Writing Methods

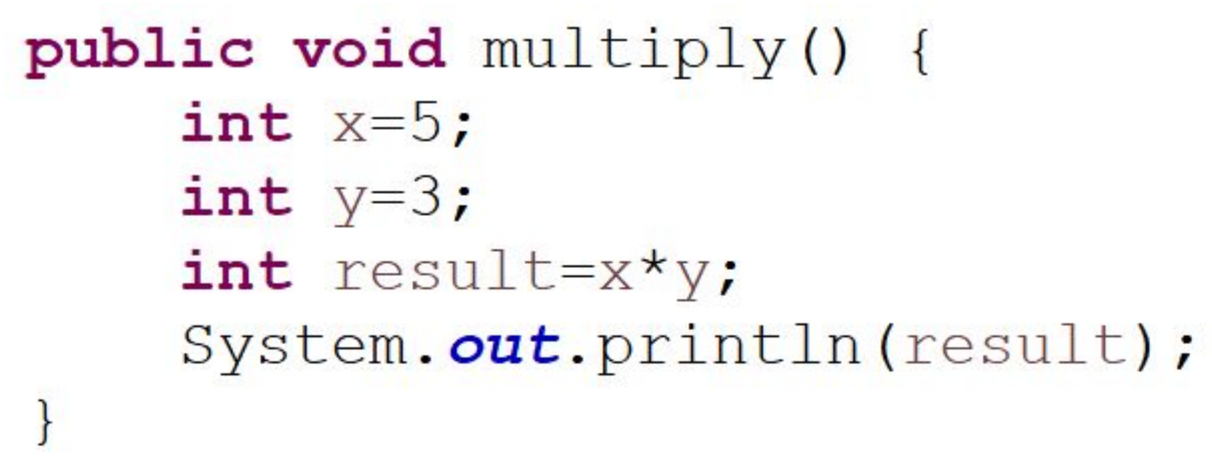
- There 2 steps in creating a method.
- The first step is method declaration which is also known as method definition or implementation. Below is an example of a method declaration/implementation. The name of the method is **multiply**. The method is **public**, and it’s **void** method, so it returns nothing. The method doesn’t take any **paramete**r because there is nothing inside the parentheses of the method. The **body** of the method is everything that is included inside the **curly braces**.



```
public void multiply() {
     int x=5;
     int y=3;
     int result=x*y;
     System.out.println(result);
}
```

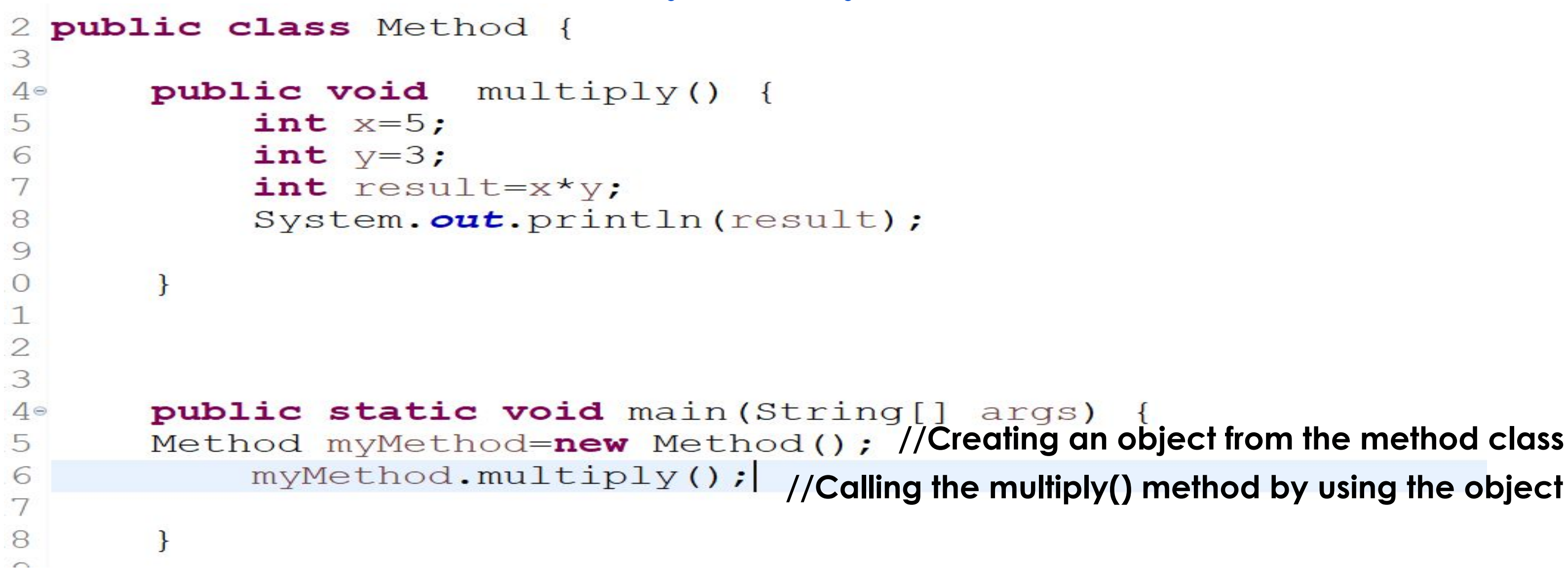
Example of a void method

- This is also an **instance(object) method** since the static keyword is not included in the method header. This method is not accessible until an object(instance) is created.

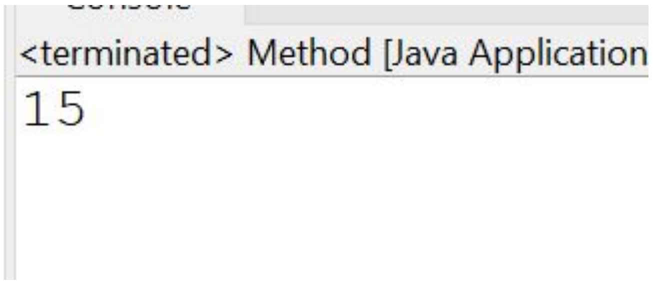


```java
public void multiply() {
     int x=5;
     int y=3;
     int result=x*y;
     System.out.println(result);
 }
```

Writing and calling a void method

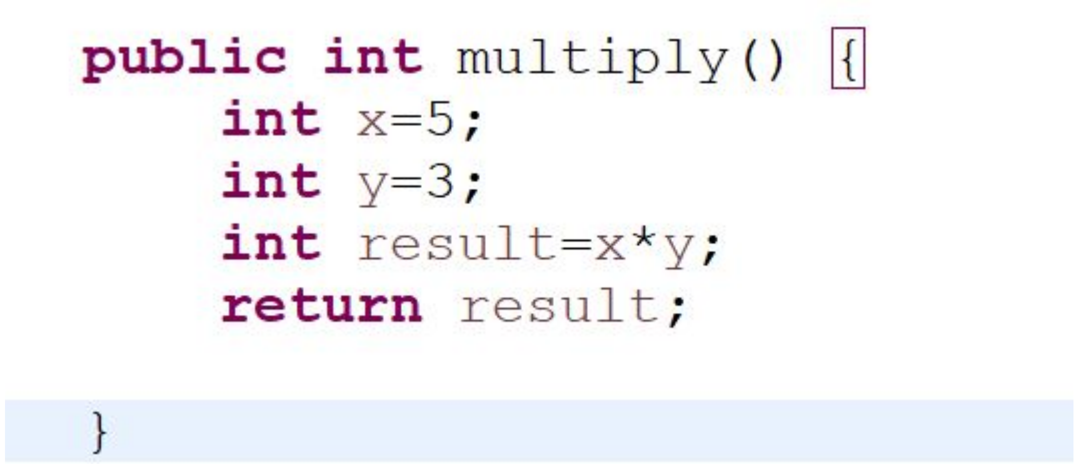


```
public class Method {

    public void  multiply() {
        int x=5;
        int y=3;
        int result=x*y;
        System.out.println(result);

    }



    public static void main(String[] args)  {
    Method myMethod=new Method(); //Creating an object from the method class
        myMethod.multiply();  //Calling the multiply() method by using the object

    }
```



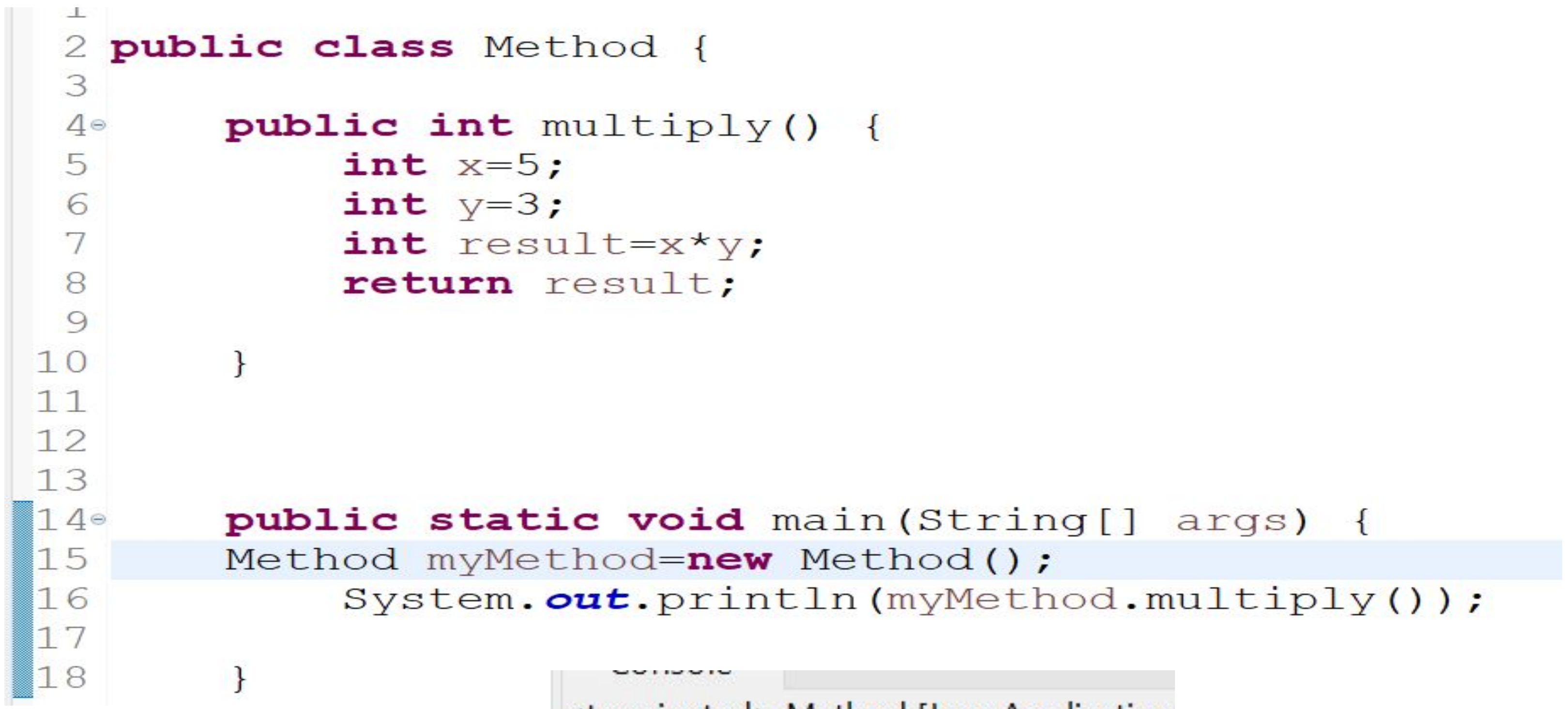
```
<terminated> Method [Java Application
15
```

Example of a non-void method

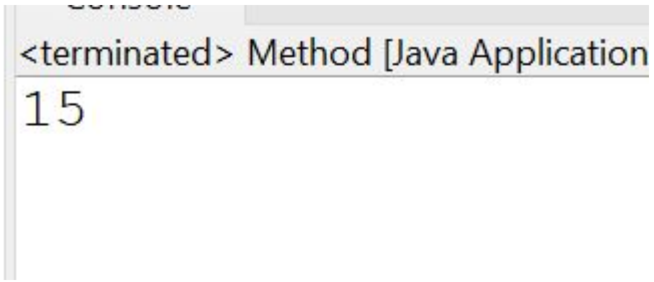


```java
public int multiply() {
    int x=5;
    int y=3;
    int result=x*y;
    return result;

}
```

Writing and calling a non-void method

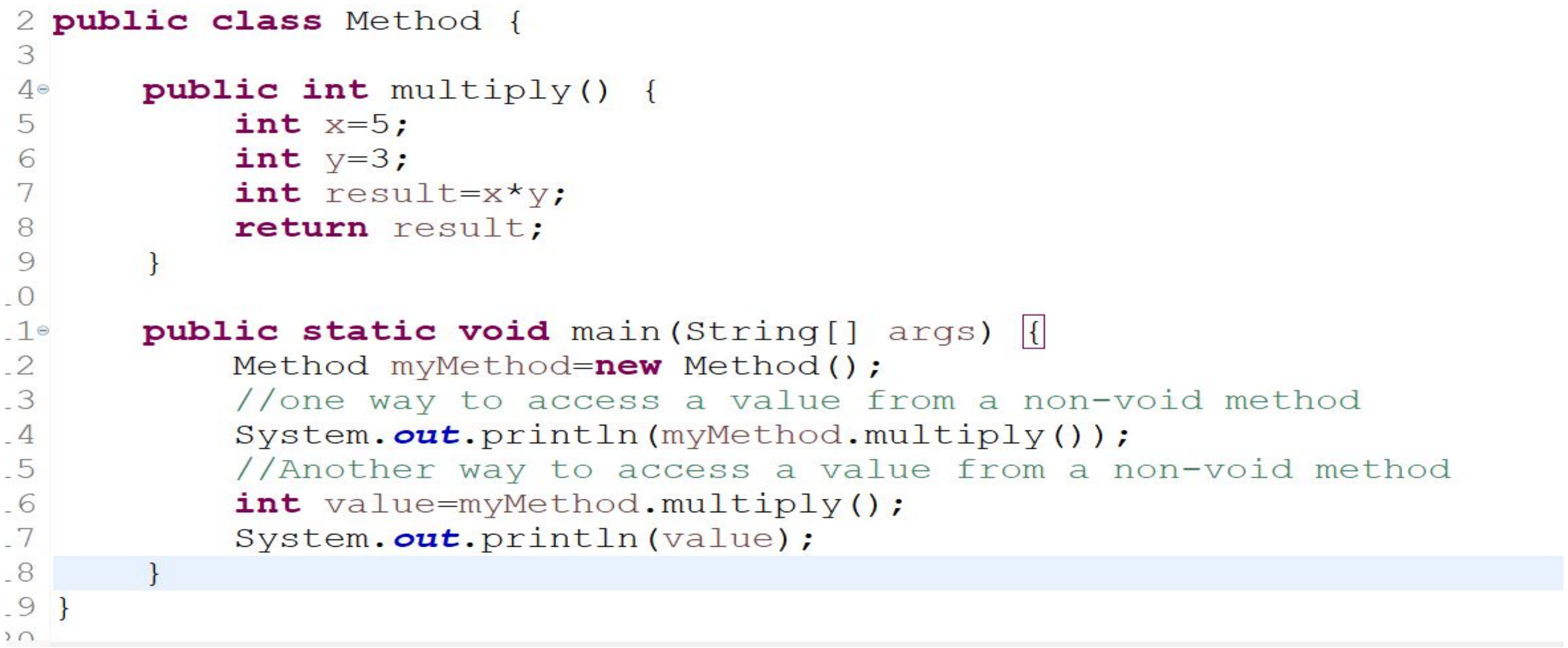


```

public class Method {

	public int multiply() {
		int x=5;
		int y=3;
		int result=x*y;
		return result;

	}



	public static void main(String[] args) {
	Method myMethod=new Method();
		System.out.println(myMethod.multiply());

	}
```

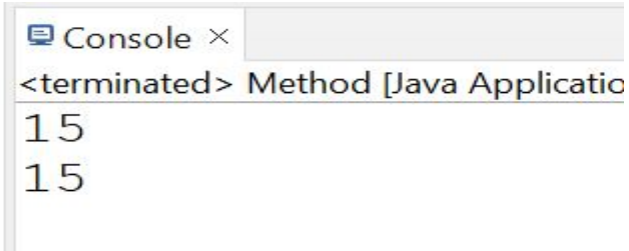


```
<terminated> Method [Java Application
15
```

Writing and calling a non-void method



```java
public class Method {

	public int multiply() {
		int x=5;
		int y=3;
		int result=x*y;
		return result;
	}

	public static void main(String[] args) {
		Method myMethod=new Method();
		//one way to access a value from a non-void method
		System.out.println(myMethod.multiply());
		//Another way to access a value from a non-void method
		int value=myMethod.multiply();
		System.out.println(value);
	}
}
```

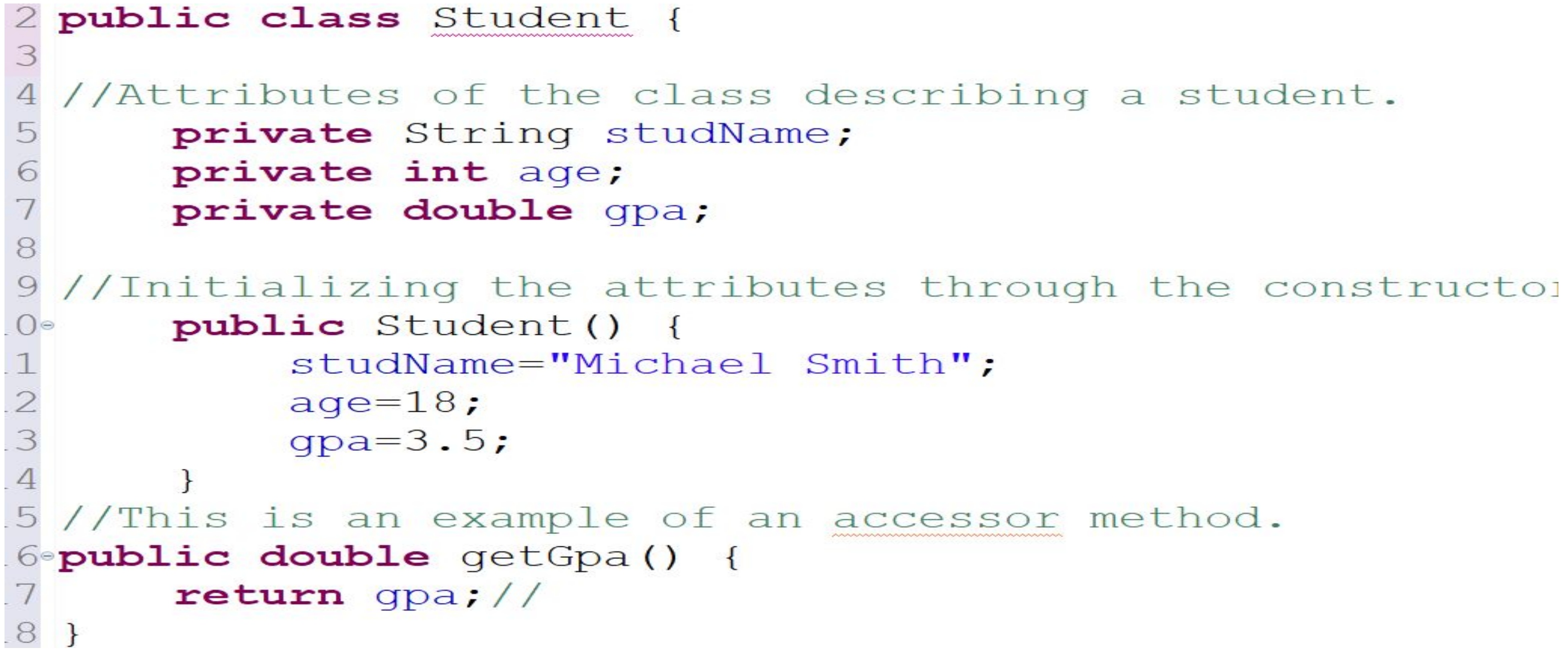


```
<terminated> Method [Java Applicatio
15
15
```

Accessor Methods

Define behaviors of an object through non-void methods without parameters written in a class.

- Accessor methods provide other objects read-only access to values of instance variables or static variables.
- It must be a non-void method that returns a single value.
- The code below shows an example of a accessor method that returns the gpa for a student.



```java
public class Student {

//Attributes of the class describing a student.
        private String studName;
        private int age;
        private double gpa;

//Initializing the attributes through the constructo
        public Student() {
                studName="Michael Smith";
                age=18;
                gpa=3.5;
        }
//This is an example of an accessor method.
public double getGpa() {
        return gpa;//
}
```

An **accessor method** is a non-void method that returns a value stored in an instance variable. Non-void methods return a value that is the same type as the return type in the method signature.

MyConsole.java

```java
public class MyConsole {
  public static void main(String args) {

    Dessert donut = new Dessert();

    System.out.print(donut.getPrice());
  }
}
```

Dessert.java

```java
public class Dessert {
  private double price;
  . . .
  public double getPrice() {
    return price;
  }
}
```

It is **public** so it is accessible from outside of the class.

The **return type** is the type of the value to be given from the method - usually matches the variable type.

The name of the method is typically **get** followed by the name of the instance variable.

The **return keyword** exits the method and provides the value to where the method is called.

Then we specify the instance variable we want to return.

An **accessor method** gives the value stored in an instance variable.

MyConsole.java

```java
public class MyConsole {
  public static void main(String args) {

    Dessert donut = new Dessert();

    System.out.print(donut.getPrice());
  }
}
```

Dessert.java

```java
public class Dessert {
  private double price;
  . . .
  public double getPrice() {
    return price;
  }
}
```

Since the method returns a value, we can either print it, store it in a variable or use it as part of an expression.

We call the accessor method on the object using the **dot** operator.

When we call the method, it will give us the value stored in the **price** instance variable.

AP Practice Question

Consider the following class definition. The class does not compile.

```
public class Bakery {
   private int numEmployees;

   // Constructor not shown

   public getNumEmployees() {
      return numEmployees;
   }
}
```

All methods (except constructors) need a return type(such as int, boolean, void). Accessor methods return a value(the state of the attribute), so it should have a return type that matches the datatype of the instance variable **numEmployees** (which is **int**).

The accessor method **getNumEmployees()** is intended to return **numEmployees**. Which of the following best explains why the class does not compile?

A. The **getNumEmployees()** method should be declared as **private**.

B. The **getNumEmployees()** method requires a parameter.

C. The return type of the **getNumEmployees()** method needs to be defined as **String**.

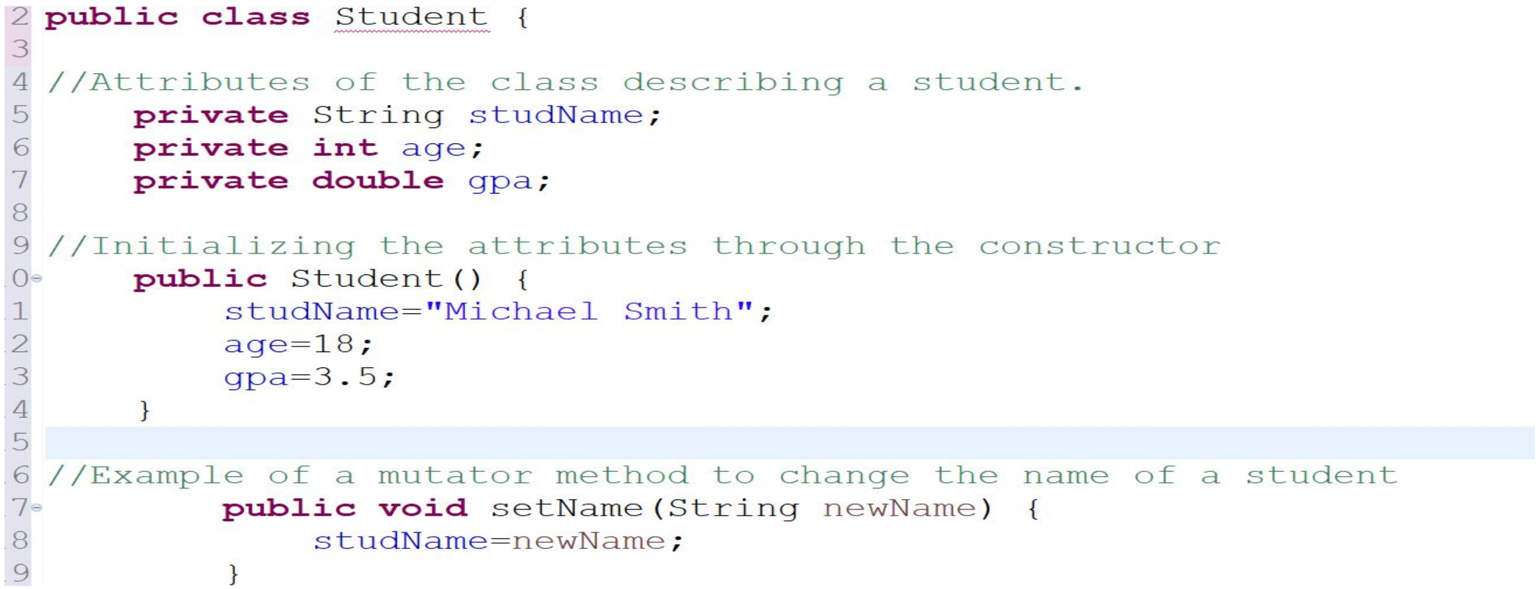
[✓ D. The return type of the **getNumEmployees()** method needs to be defined as **int**.]

E. The return type of the **getNumEmployees()** method needs to be defined as **void**.

Mutator Methods

Define behaviors of an object through void methods with or without parameters written in a class.

- A mutator method is usually created with the **void** keyboard being placed before the method name in the method signature
- Mutator methods are void methods that modify the values of instance and static variables.
- Unlike accessor methods, mutator methods do not return a value.
- The code below shows an example of a mutator method that changes the current name of of a student.



```
public class Student {

//Attributes of the class describing a student.
    private String studName;
    private int age;
    private double gpa;

//Initializing the attributes through the constructor
    public Student() {
        studName="Michael Smith";
        age=18;
        gpa=3.5;
    }

//Example of a mutator method to change the name of a student
        public void setName(String newName) {
            studName=newName;
        }
```

A **mutator method** changes the value stored in an instance variable.

MyConsole.java

```java
public class MyConsole {
  public static void main(String args) {

     Dessert donut = new Dessert();

    donut.setPrice(2.99);
  }
}
```

Dessert.java

```java
public class Dessert {
  private double price;
  . . .
  public void setPrice(double newPrice) {
    price = newPrice;
  }
}
```

It is **public** so it is accessible from outside of the class.

The return type is **void** since it does not return a value.

The name of the method is typically **set** followed by the name of the instance variable.

We specify a parameter that matches the type of the instance variable.

Then we assign the value passed to the parameter to the instance variable.

A **mutator method** changes the value stored in an instance variable.

MyConsole.java

```
public class MyConsole {
  public static void main(String args) {

    Dessert donut = new Dessert();

    donut.setPrice(2.99);
  }
}
```

Dessert.java

```
public class Dessert {
  private double price;
  . . .
  public void setPrice(double newPrice) {
    price = newPrice;
  }
}
```

As void methods, they don't return any information they simply perform an action, like changing the state of an object.

When we call the method, we pass the new value to set for the **`price`** instance variable.

AP Practice Question

Consider the following class definition:

```
public class Liquid
{
    private int currentTemp;

    public Liquid(int temp)
    {
        currentTemp = temp;
    }

    public void resetTemp()
    {
        currentTemp = newTemp;
    }
}
```

Which of the following best identifies the reason why the class does not compile?

(A) The constructor method does not have a header

(B) the **`resetTemp`** method is missing a return type

(C) the constructor should not have a parameter

[✓ (D) the **`resetTemp`** method should have a parameter]

(E) the instance variable **`currentTemp`** should be public instead of private

3.6 Passing and Returning References of an Object

Learning Objectives:

- Develop code to define behaviors of an object through methods written in a class using object references and determine the result of calling these methods.

Key Terms to Know

Passing and Returning References of an Object in Java

- in Java, **objects are passed by value of the reference**, meaning the reference (or memory address) to the object is passed **by value**, not the actual object itself. This allows methods to access and modify the object's internal state but not change the reference itself in the caller.

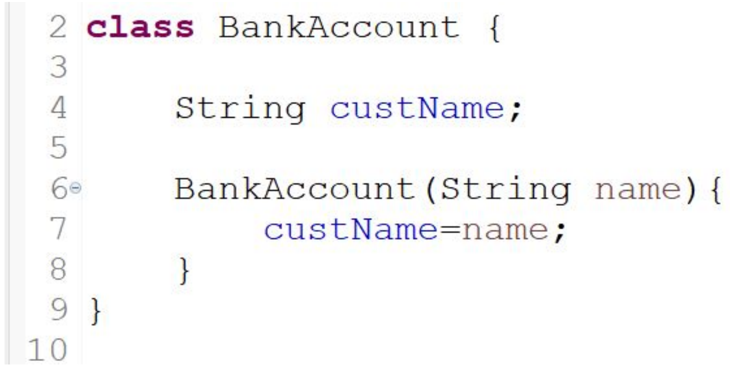
Passing Object References to Methods (Ex1)

```java
public class BankDemo {

 // Method receives an object reference
 public static void addBonus(BankAccount account) {
     account.deposit(500);  // Modifies the original object
 }

public static void main(String[] args) {
    BankAccount myAccount = new BankAccount("Alice", 1000);

    System.out.println("Before:");
    myAccount.displayInfo();

    // Pass the object reference to the method
    addBonus(myAccount);

    System.out.println("After adding bonus:");
    myAccount.displayInfo();
}

 }
```

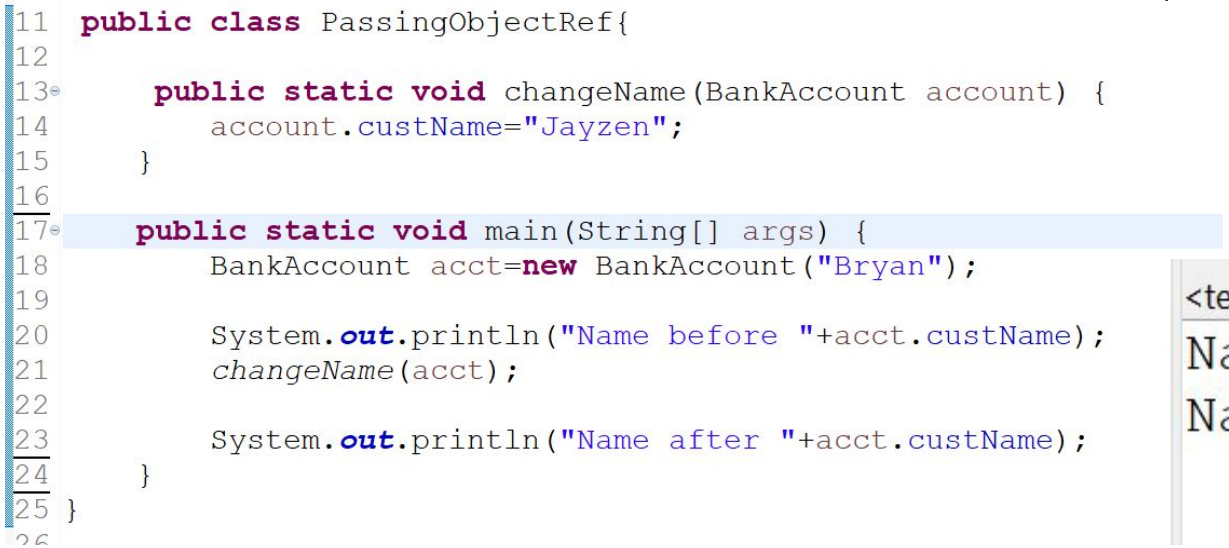
```
Before:
Alice has $1000.0
After adding bonus:
Alice has $1500.0
```

- The method `addBonus(BankAccount account)` receives a **reference** to `myAccount`.
- It calls `account.deposit(500)`, which modifies the original `myAccount` object.
- This shows that the object reference, not a copy, is passed.

Passing Object References to Methods (Ex.2)

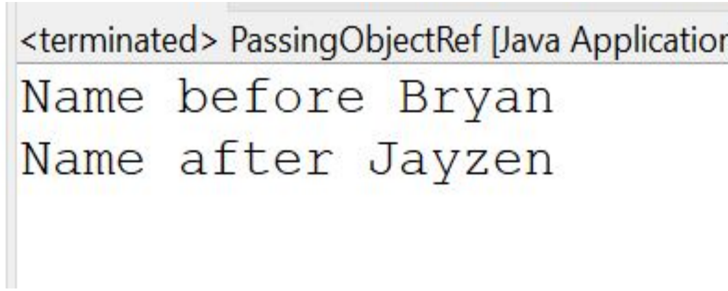


```java
class BankAccount {

    String custName;

    BankAccount(String name){
        custName=name;
    }
}

```



```java
public class PassingObjectRef{

    public static void changeName(BankAccount account) {
        account.custName="Jayzen";
    }

    public static void main(String[] args) {
        BankAccount acct=new BankAccount("Bryan");

        System.out.println("Name before "+acct.custName);
        changeName(acct);

        System.out.println("Name after "+acct.custName);
    }
}
```

The changeName method changes the name from the BankAccount object. Since the method received the reference (**a copy of the reference**), it affects the actual object.

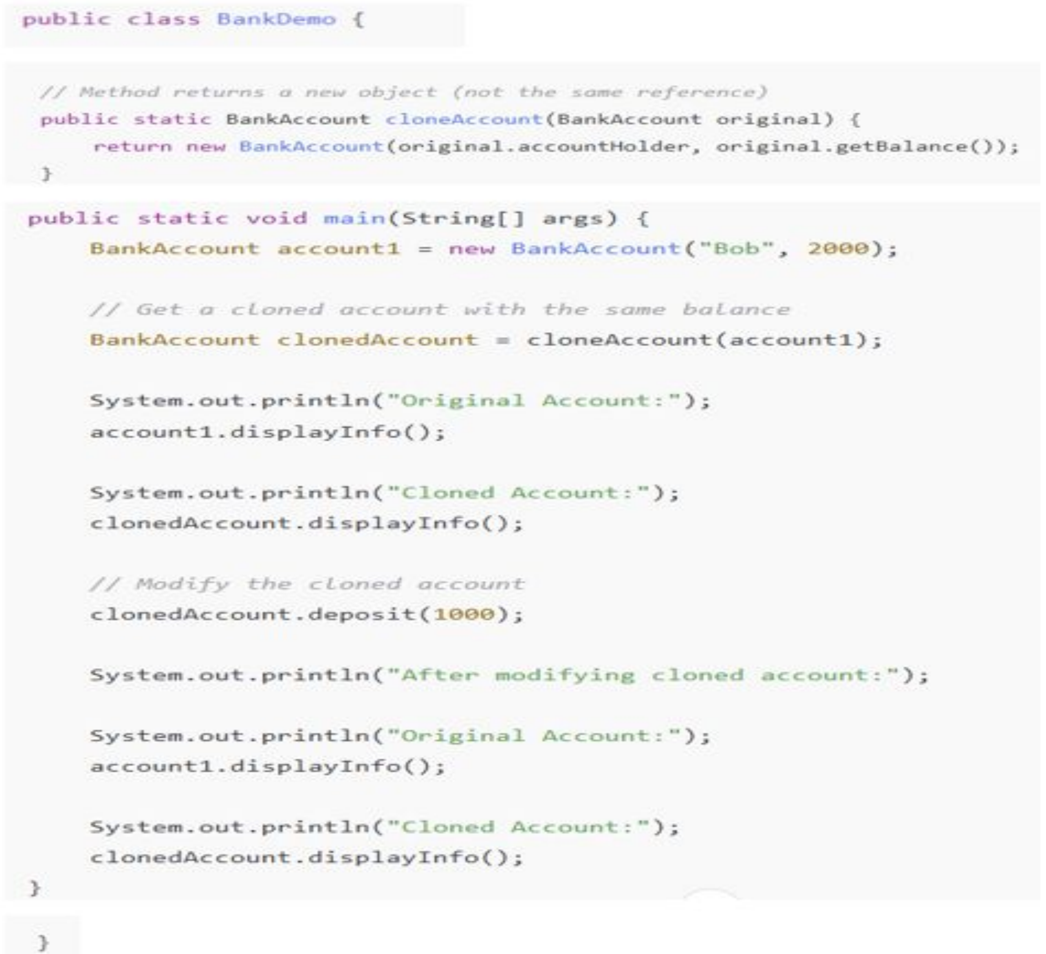


```
<terminated> PassingObjectRef [Java Applicatio
Name before Bryan
Name after Jayzen
```

Returning Object References from Methods

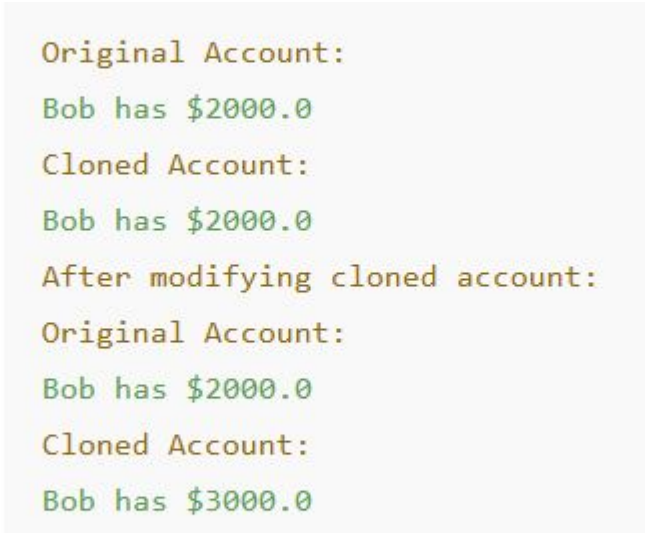
- A method can return an object reference, which means the caller receives the reference to the same object (or a new object, depending on how it's created).
- When an object is passed to a method, the method receives a **copy of the reference** to the object.
- Changes to the object's fields inside the method will affect the original object.

Passing Object References to Methods (Examples1)



```
public class BankDemo {

 // Method returns a new object (not the same reference)
 public static BankAccount cloneAccount(BankAccount original) {
     return new BankAccount(original.accountHolder, original.getBalance());
 }

public static void main(String[] args) {
    BankAccount account1 = new BankAccount("Bob", 2000);

    // Get a cloned account with the same balance
    BankAccount clonedAccount = cloneAccount(account1);

    System.out.println("Original Account:");
    account1.displayInfo();

    System.out.println("Cloned Account:");
    clonedAccount.displayInfo();

    // Modify the cloned account
    clonedAccount.deposit(1000);

    System.out.println("After modifying cloned account:");

    System.out.println("Original Account:");
    account1.displayInfo();

    System.out.println("Cloned Account:");
    clonedAccount.displayInfo();
}

 }
```

Result:



```
Original Account:
Bob has $2000.0
Cloned Account:
Bob has $2000.0
After modifying cloned account:
Original Account:
Bob has $2000.0
Cloned Account:
Bob has $3000.0
```

The method `cloneAccount()` returns a **new BankAccount object**, not the original reference.

Changes to `clonedAccount` do not affect `account1`.

Where the Reference Itself Doesn't Change:

While the object’s fields can be modified when passed to a method, assigning a new object to the parameter inside the method does **not affect** the original reference.

Example Where the Reference Itself Doesn't Change:

```
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }
}

public class ReferenceTest {
    public static void changeReference(Student s) {
        s = new Student("Bob");
    }

    public static void main(String[] args) {
        Student student = new Student("Eve");
        System.out.println("Before: " + student.name);

        changeReference(student);

        System.out.println("After: " + student.name);
    }
}
```

Before: Eve
After: Eve

The `changeReference` method creates a new `Student` object, but this only changes the local copy of the reference inside the method. The original reference in `main` remains unchanged.

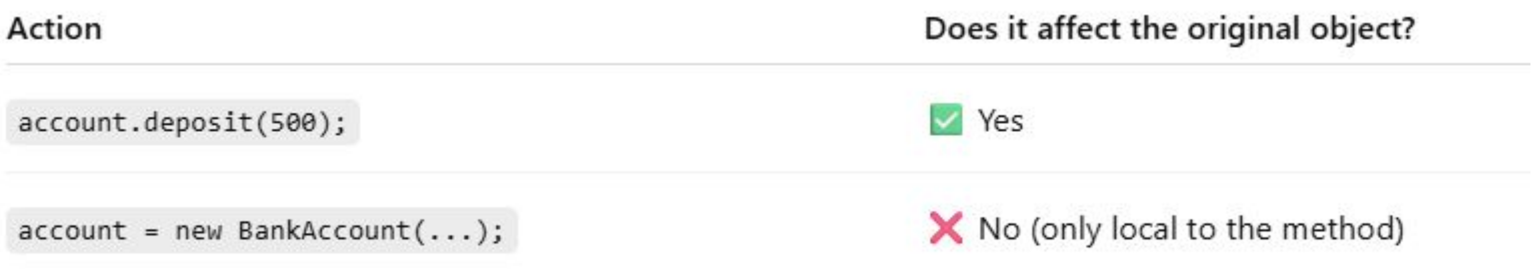
Summary of Object References in Java

Java passes **object references by value**. This means the reference (like an address) is copied, but both the caller and the callee refer to the **same object**.

If a method modifies the object (like changing balance), it affects the original.

If a method reassigns the object reference itself (like `account = new BankAccount()` inside the method), it does not affect the original reference outside the method.

When `account = new BankAccount("Charlie", 5000);//` is executed, it makes the **local copy**
// of the reference point to a new object.



Action	Does it affect the original object?
`account.deposit(500);`	✅ Yes
`account = new BankAccount(...);`	❌ No (only local to the method)

3.7 Class Variables and Methods

Learning Objectives:

- Develop code to define the behaviors of a class through class methods
- Develop code to declare the class variables that belong to the class.

Class Methods (Things you need to know)

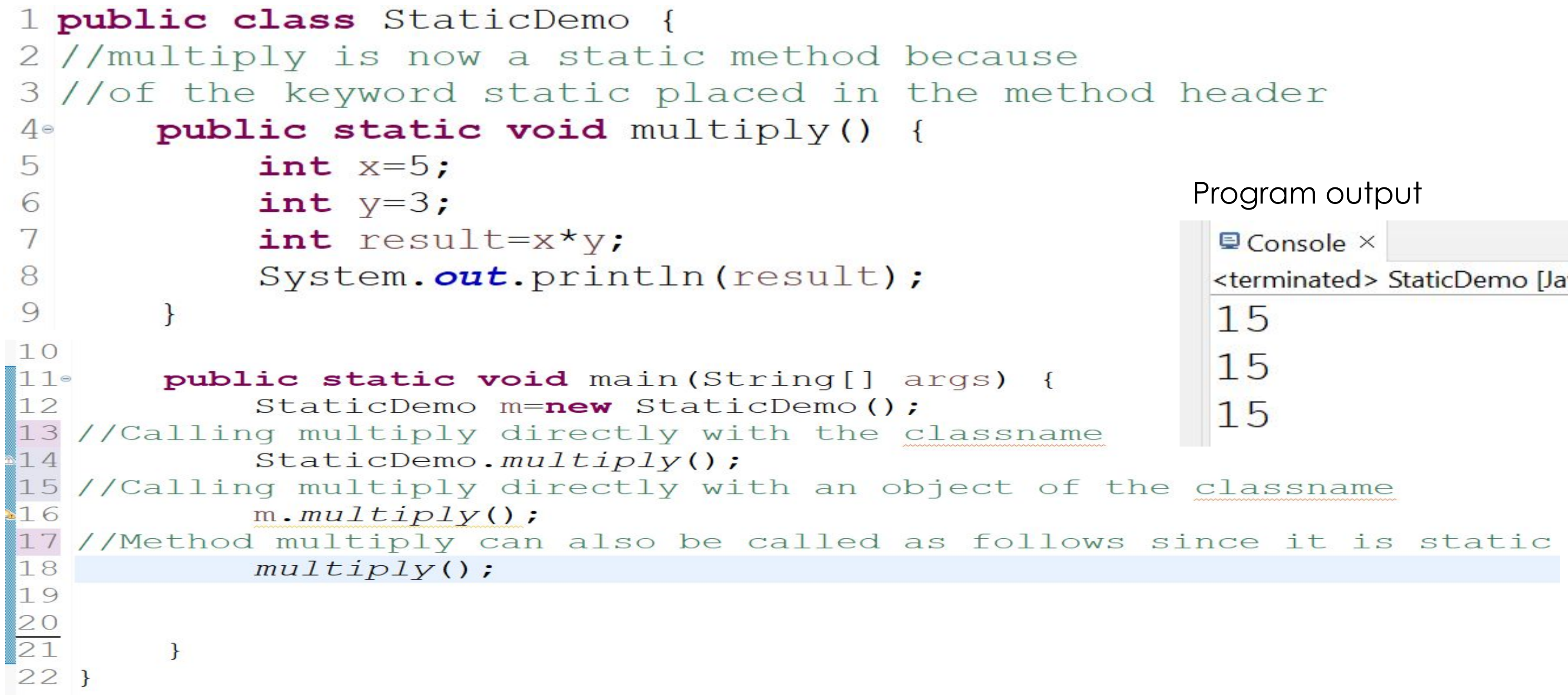
Define behaviors of a class through static methods.

- **Class methods** are also called static methods because they are connected to the class, not the object of the class.
- The keyword static needs to be included in the method header
- Class methods cannot access or change the values of instance variables or call instance methods without being passed an instance of the class via a parameter.
- Class methods can access or change the values of class variables and can call other class methods.
- Class variables are also called static variables because they are attached to the class, not the object of the class.
- Class variables belong to the clas, with all objects of a class sharing a single copy of the class variables.
- **Class variables** are declared by including the static keyword before the variable type.
- Class variables that are designated as public are accessible outside of the class by using the class name and the **dot operator**(**.**) , since they are associated with a class, not objects of a class.
- When a variable is declare as **final**, its value cannot be modified.

Class Methods (Example)

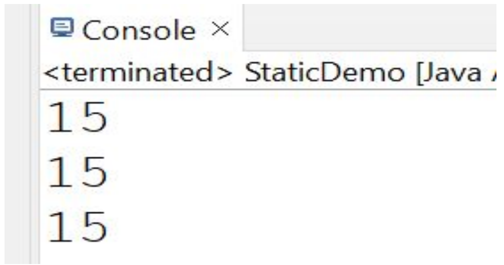
Define behaviors of a class through static methods.

- Here is an example of a static method.



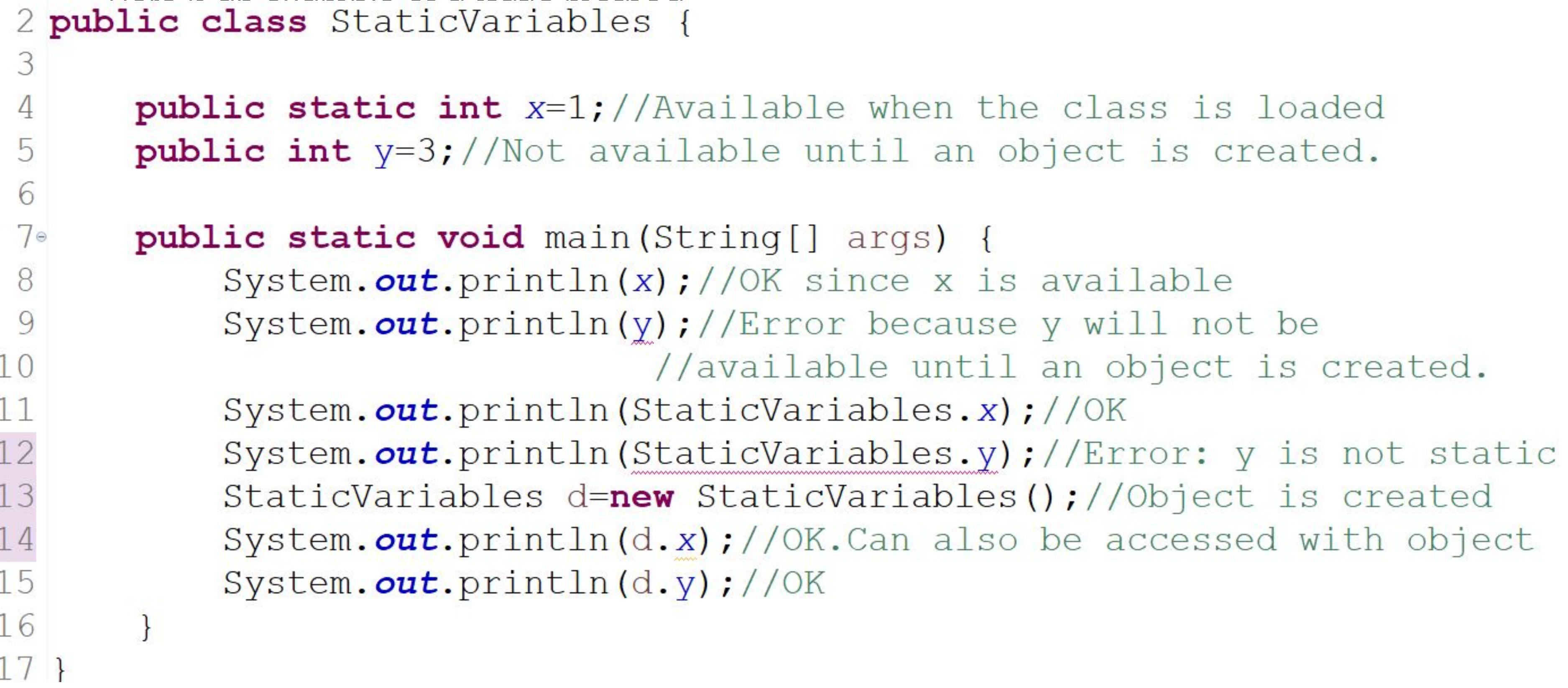
```java
public class StaticDemo {
//multiply is now a static method because
//of the keyword static placed in the method header
    public static void multiply() {
        int x=5;
        int y=3;
        int result=x*y;
        System.out.println(result);
    }

    public static void main(String[] args) {
        StaticDemo m=new StaticDemo();
//Calling multiply directly with the classname
        StaticDemo.multiply();
//Calling multiply directly with an object of the classname
        m.multiply();
//Method multiply can also be called as follows since it is static
        multiply();


    }
}
```

Program output



```
<terminated> StaticDemo [Java
15
15
15
```

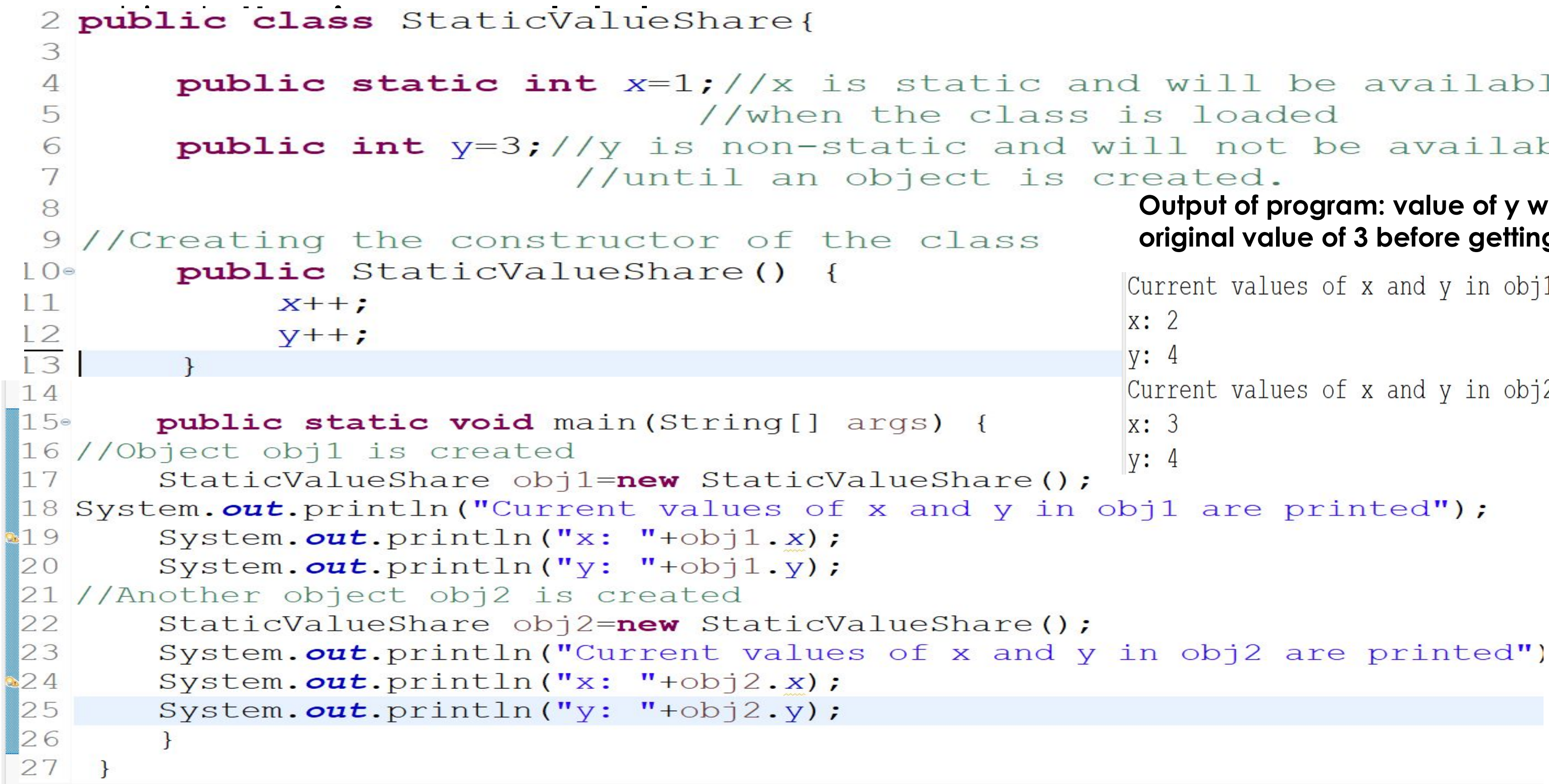
Class Variables (Examples)



```java
public class StaticVariables {

    public static int x=1;//Available when the class is loaded
    public int y=3;//Not available until an object is created.

    public static void main(String[] args) {
        System.out.println(x);//OK since x is available
        System.out.println(y);//Error because y will not be
                            //available until an object is created.
        System.out.println(StaticVariables.x);//OK
        System.out.println(StaticVariables.y);//Error: y is not static
        StaticVariables d=new StaticVariables();//Object is created
        System.out.println(d.x);//OK.Can also be accessed with object
        System.out.println(d.y);//OK
    }
}
```

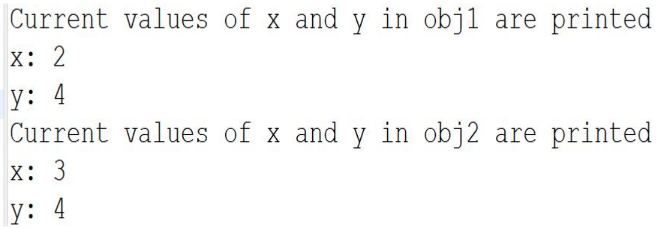
Static variables are shared by all objects(instances) of the class

- Unlike non-static variables, static variables are shared by all objects(**instances**) of the class. Any change made to a static variable by one object will be carried over to the next



```java
public class StaticValueShare{

	public static int x=1;//x is static and will be available
	                          //when the class is loaded
	public int y=3;//y is non-static and will not be available
	                     //until an object is created.

//Creating the constructor of the class
	public StaticValueShare()  {
		x++;
		y++;
	}

	public static void main(String[] args)  {
//Object obj1 is created
	StaticValueShare obj1=new StaticValueShare();
System.out.println("Current values of x and y in obj1 are printed");
	System.out.println("x: "+obj1.x);
	System.out.println("y: "+obj1.y);
//Another object obj2 is created
	StaticValueShare obj2=new StaticValueShare();
	System.out.println("Current values of x and y in obj2 are printed")
	System.out.println("x: "+obj2.x);
	System.out.println("y: "+obj2.y);
	}
 }
```

Output of program: value of y was reset to original value of 3 before getting in obj2.



```
Current values of x and y in obj1 are printed
x: 2
y: 4
Current values of x and y in obj2 are printed
x: 3
y: 4
```

3.8 Scope and Access

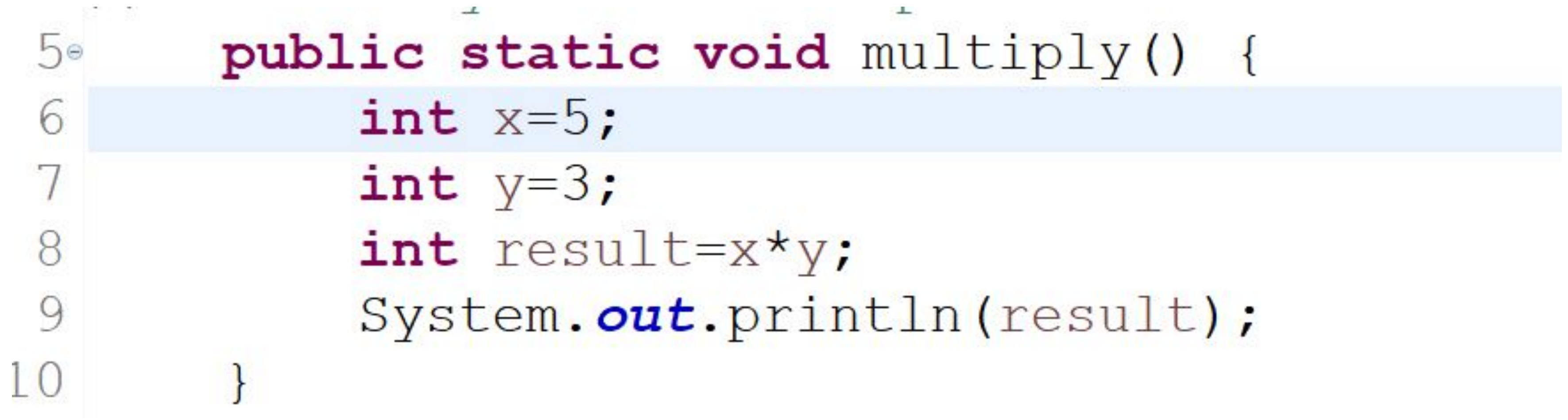
Learning Objectives:

Explain where variables can be used in the program code.

- Scope of variables determines where the variable can be accessed.
- **Local variables** are only accessible where they are declared.
- Variables declared as formal parameters and those declared inside a method or constructor are accessible only inside that method or constructor.
- When a local variable has the same name as an **instance variable**, the local variable will take precedence instead of the instance variable.
- Variables are often used by programmers to generalize solutions to a problem since it allows different input to be used.
- Through the use of method decomposition, programmers often use variables which help them break down big problems into smaller, manageable that can be handled individually.

Variable Scope and Access(Declared in the body of method)

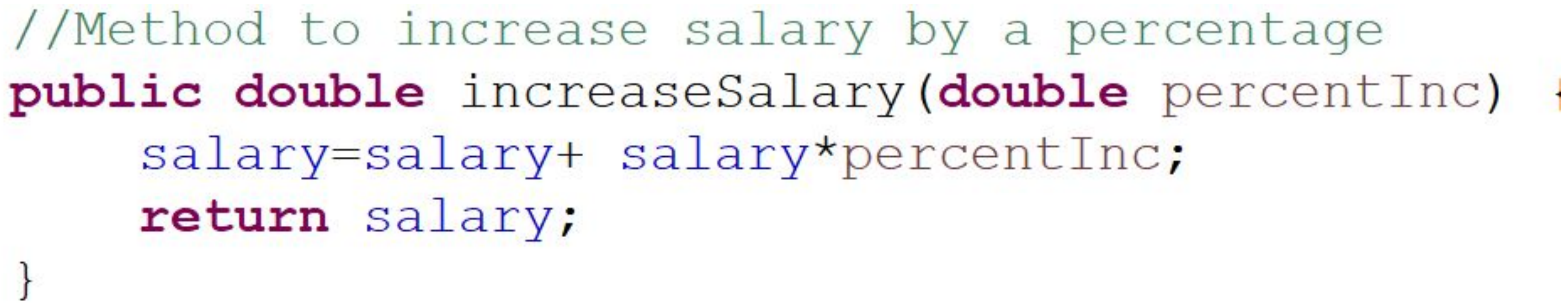
In the example below, x and y are local variables because they are declared inside the body of a method.



```
    public static void multiply() {
        int x=5;
        int y=3;
        int result=x*y;
        System.out.println(result);
    }
```

Variable Scope and Access(Declared as a parameter in the method header)

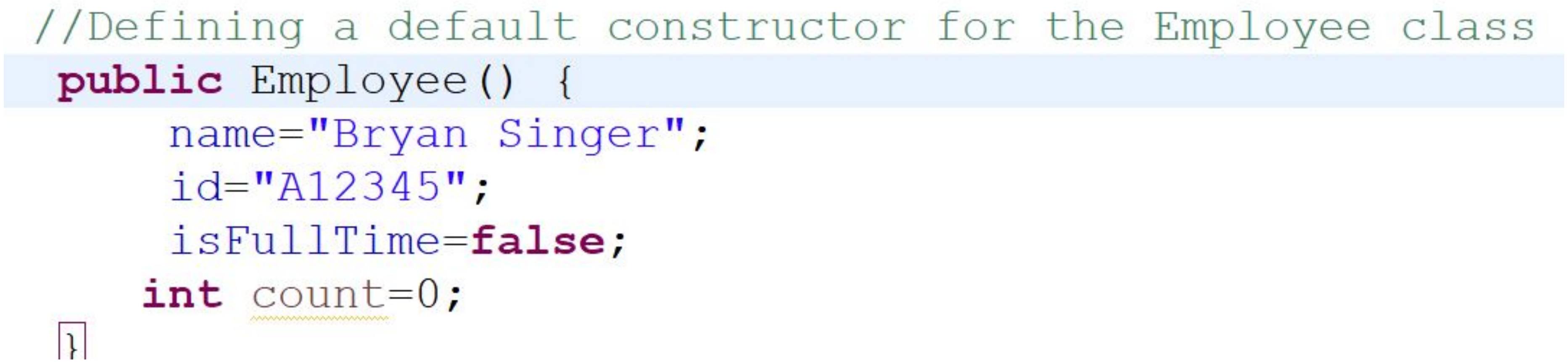
In the example below, **percentInc** is known as formal parameter which is local and accessible only within the body of the **increaseSalary** method.



```
//Method to increase salary by a percentage
public double increaseSalary(double percentInc)
    salary=salary+ salary*percentInc;
    return salary;
}
```

Variable Scope and Access (Declared in the body of a constructor)

In the example below, **count** is a local variable because it is declared inside a constructor called **Employee**, so count is accessible only within the body of the constructor.



```
//Defining a default constructor for the Employee class
 public Employee() {
     name="Bryan Singer";
     id="A12345";
     isFullTime=false;
    int count=0;
 }
```

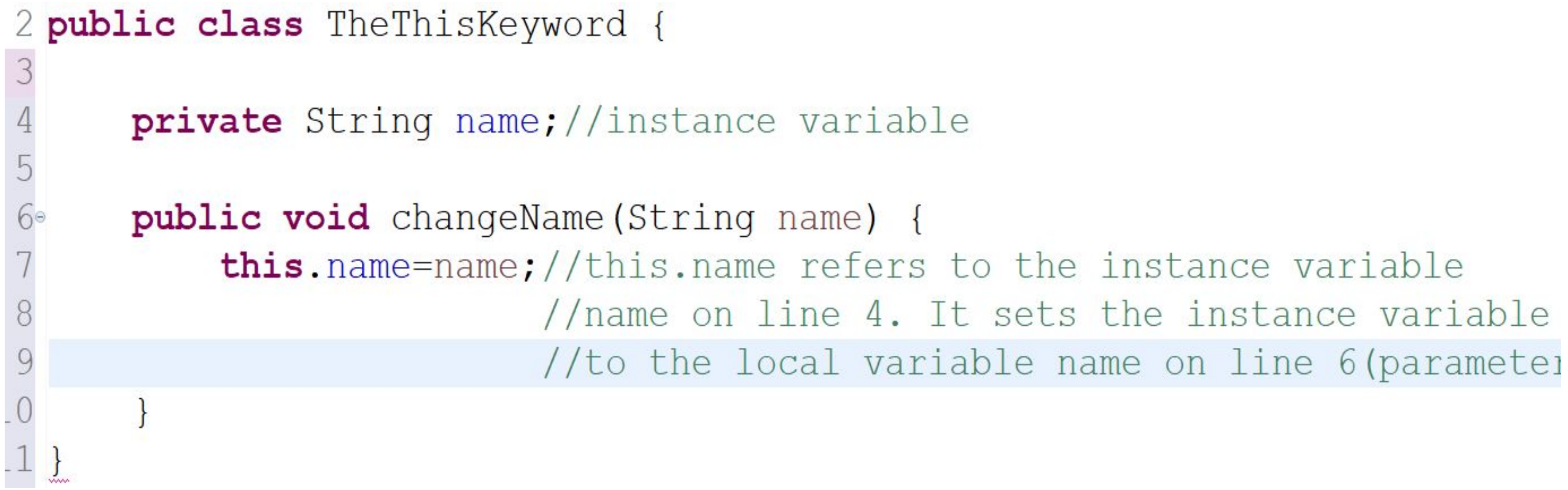
3.9 The this Keyword

Learning Objectives:

Develop code for expressions that are self-referencing and determine the result of these expressions.

- The **this** keyword is used inside a non-static method or constructor to refer to the object whose method or constructor is being executed.
- Class methods do not have a this reference
- When a method is invoked(called), the **this** keyword can be passed as an argument to that method to refer to the current object.
    - Below are some examples of how the **this** keyword is used with a method and constructor.

When a local variable has the same name as an instance variable.



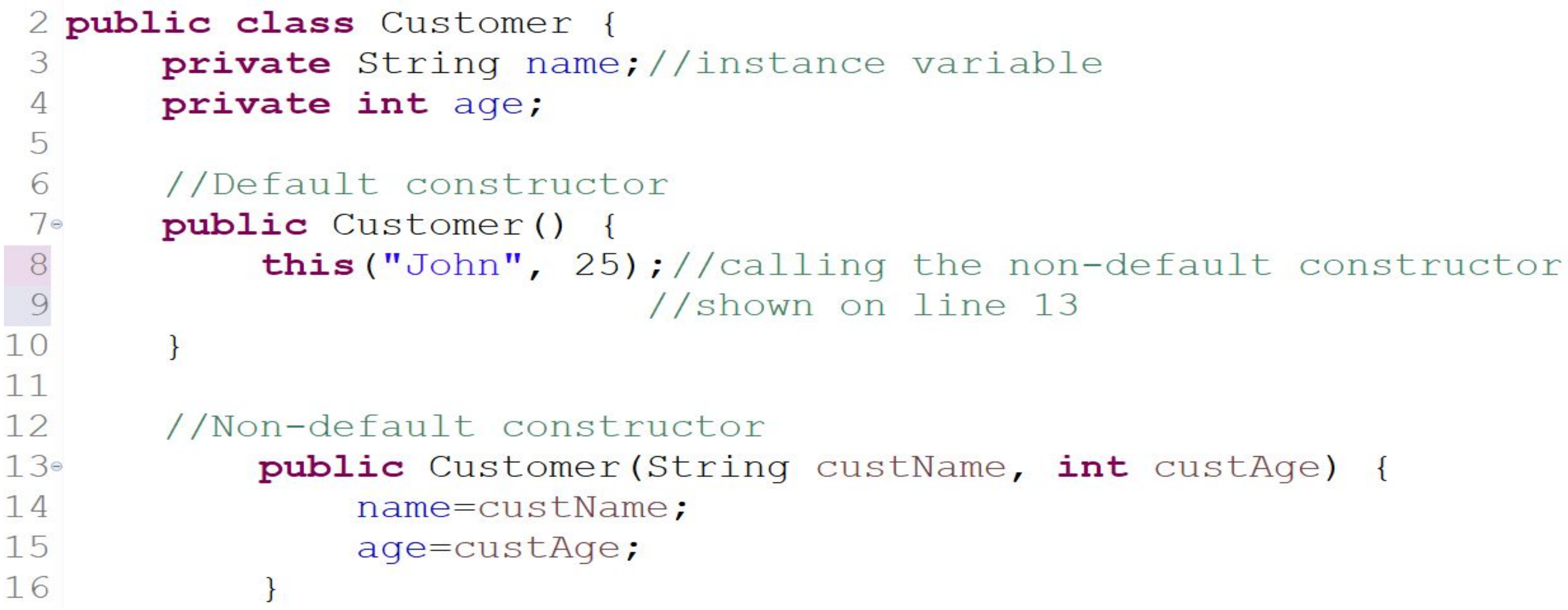
```
 2 public class TheThisKeyword {
 3
 4     private String name;//instance variable
 5
 6     public void changeName(String name) {
 7         this.name=name;//this.name refers to the instance variable
 8                        //name on line 4. It sets the instance variable
 9                        //to the local variable name on line 6(paramete
10     }
11 }
```

The this Keyword

Learning Objectives:

Evaluate object reference expressions that use the keyword this.

- Below are some examples of how the **this** keyword is used with a method and constructor.



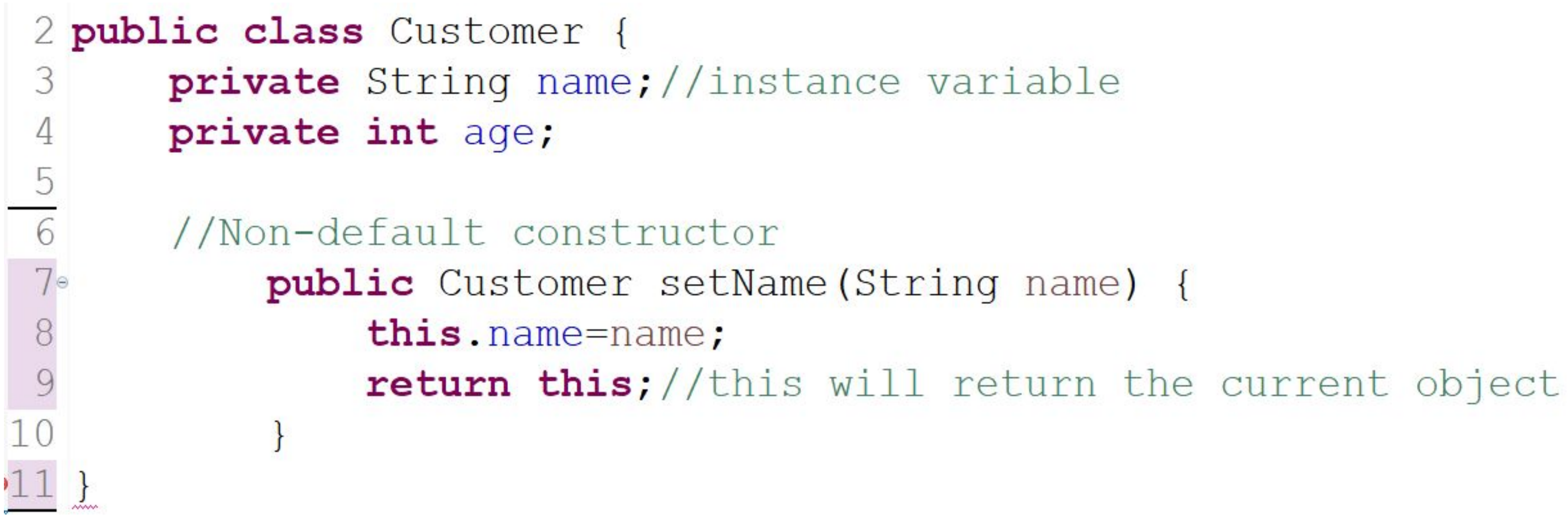
```
 2 public class Customer {
 3     private String name;//instance variable
 4     private int age;
 5
 6     //Default constructor
 7     public Customer() {
 8         this("John", 25);//calling the non-default constructor
 9                         //shown on line 13
10     }
11
12     //Non-default constructor
13         public Customer(String custName, int custAge) {
14             name=custName;
15             age=custAge;
16         }
```

The this Keyword

Learning Objectives:

Evaluate object reference expressions that use the keyword this.

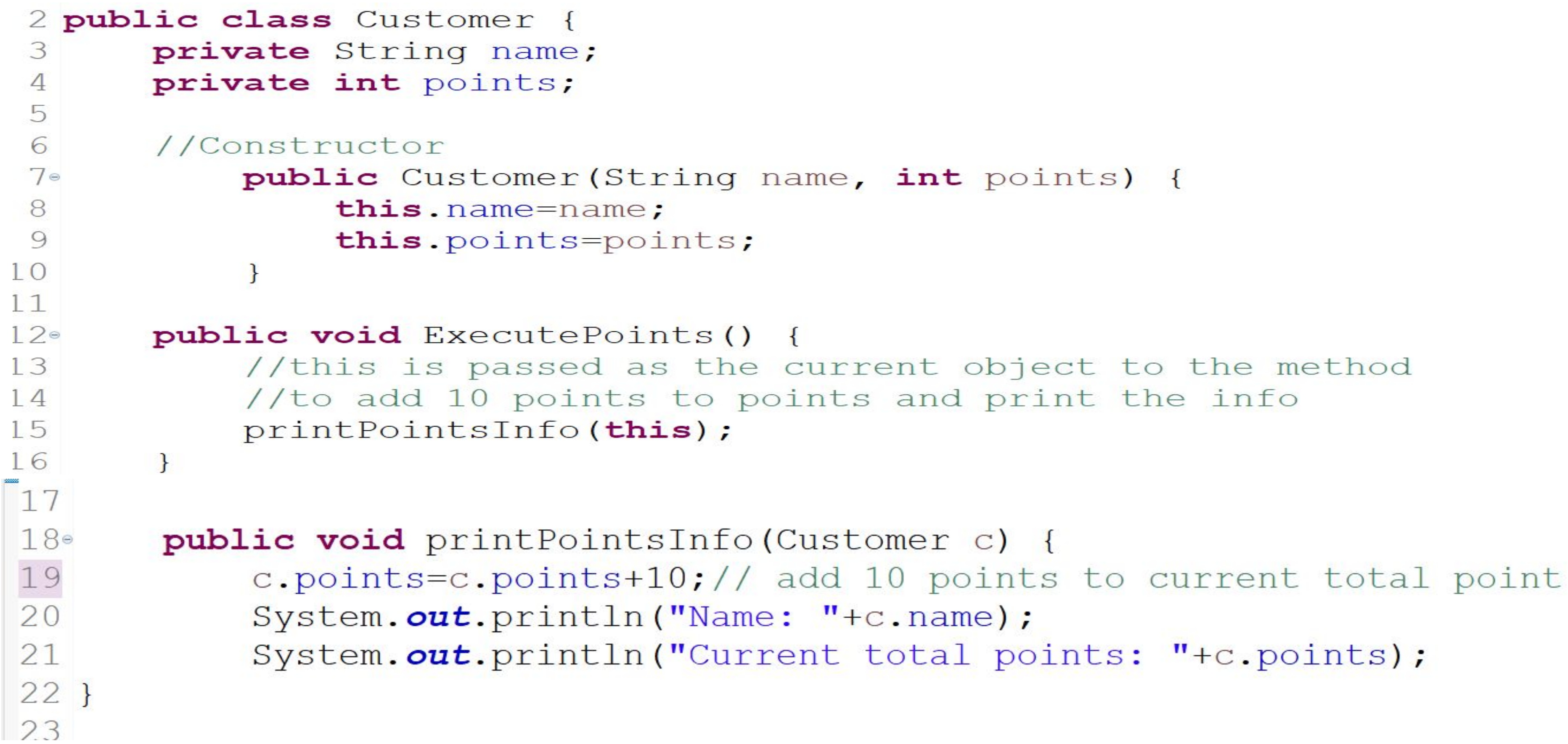
- Below are some examples of how the **this** keyword is used with a method and constructor.



```java
public class Customer {
    private String name;//instance variable
    private int age;

    //Non-default constructor
        public Customer setName(String name) {
            this.name=name;
            return this;//this will return the current object
        }
}
```

The this Keyword

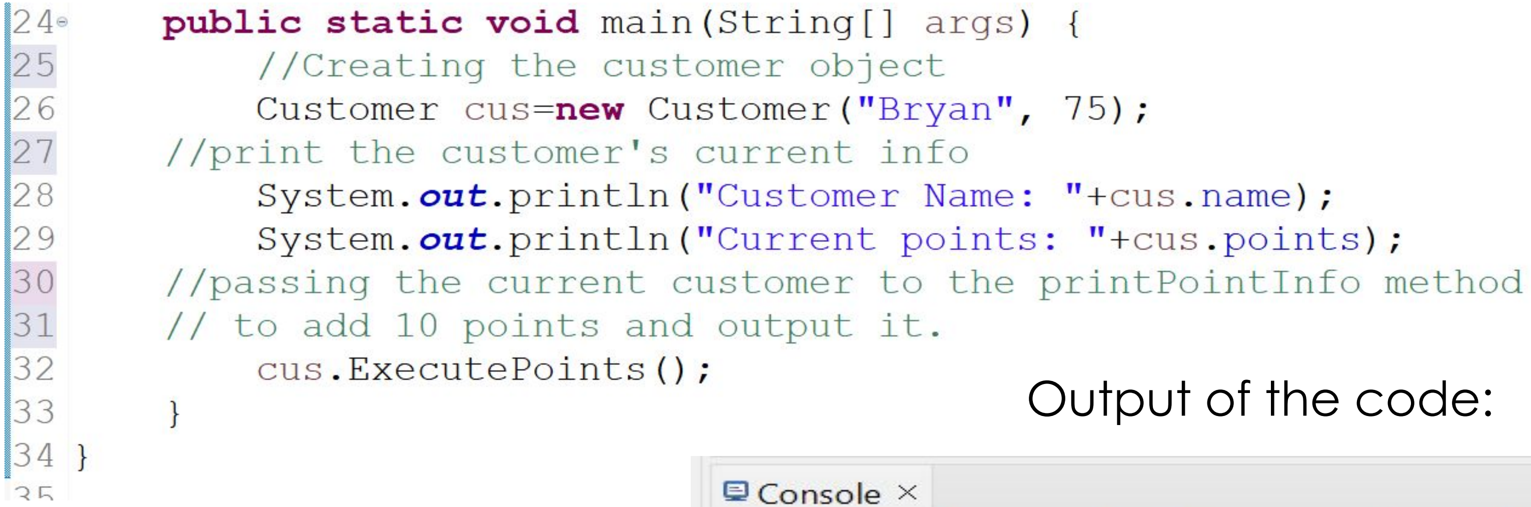
Below are some examples of how the **this** keyword is used with a method and constructor.



```java
public class Customer {
    private String name;
    private int points;

    //Constructor
        public Customer(String name, int points) {
            this.name=name;
            this.points=points;
        }

    public void ExecutePoints() {
        //this is passed as the current object to the method
        //to add 10 points to points and print the info
        printPointsInfo(this);
    }

    public void printPointsInfo(Customer c) {
        c.points=c.points+10;// add 10 points to current total point
        System.out.println("Name: "+c.name);
        System.out.println("Current total points: "+c.points);
}
```

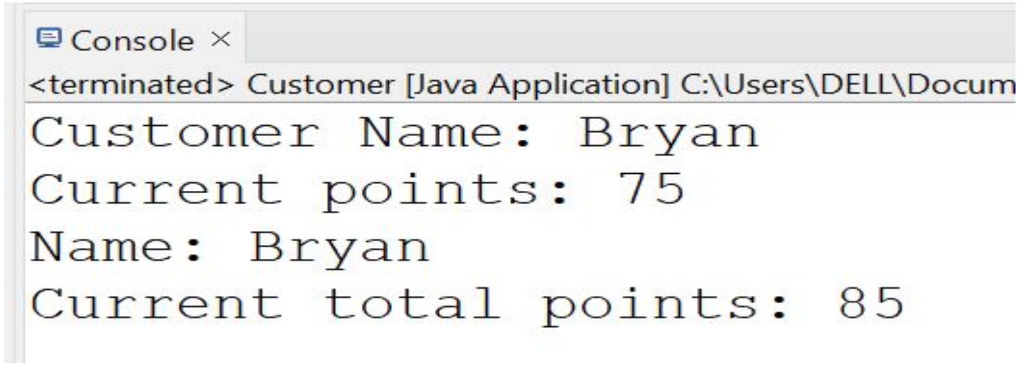
The this Keyword

Below is the testing part of the program from the previous slide



```java
    public static void main(String[] args) {
        //Creating the customer object
        Customer cus=new Customer("Bryan", 75);
    //print the customer's current info
        System.out.println("Customer Name:  "+cus.name);
        System.out.println("Current points:  "+cus.points);
    //passing the current customer to the printPointInfo method
    // to add 10 points and output it.
        cus.ExecutePoints();
    }
}
```

Output of the code:



```
<terminated> Customer [Java Application] C:\Users\DELL\Docum
Customer Name:  Bryan
Current points:  75
Name:  Bryan
Current total points:  85
```

References:

- This training document was prepared using some of the resources from the CollegeBoard, including their AP Computer Science A course page which can be found at the link below:
- https://apcentral.collegeboard.org/courses/ap-computer-science-a?utm_source=chatgpt.com
-