Unit 5: **Object-Oriented Concepts – Defining Classes**

Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U5: Object-Oriented Concepts – Defining Classes

- UML Class Diagram Representation
- Defining Java Classes
  - Class Declaration
  - Fields, Constructors, Methods
  - Constructor - Initializing Objects
  - More Complicated Java Classes
- Class Fields/Methods vs. Instance Fields/Methods
- Method Calling
  - Pass by Value

UML Class Diagram Representation

- When modeling our real world in an object-oriented approach, we create many *objects*
  - In particular many objects of the same kind are defined using a common type *class*
  - Standard classes (e.g. `String, Math, Date, JOptionPane`) may not meet all our needs for specific applications

- We need to define our own classes for our specific applications.
  - Before we code any Java class, it is important to design our classes before implementation, and communicate with others using certain ways
  - Unified Modeling Language (UML) is in particular a widely-used graphical scheme for modeling object-oriented systems

- **Unified Modeling Language** (**UML**) is a standardized general-purpose modeling language in object-oriented software engineering
  - UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems

UML Class Diagram Representation

When designing / modeling components of software system in an Object-Oriented approach before coding and implementation, we need to consider:

- How to represent the object:- What attributes / properties / states and their types the specific component (as a class of objects) should contain? … **Fields**

- How is the object behaving (behaviors):- What can the component do if it is called, what the input *parameters* is required, and what *return* when finished? … **Methods**

- How to create a new object:- how is the component newly created / instantiated, what proper input required (*parameters*)? … **Constructors**

UML Class Diagram Representation

Depending on different representations and modeling requirements, different software systems may model the same item differently. E.g.

- A student object in an academic course system should contain ***fields*** for `sID, sName` etc. *However, a student object may contain fields for representing `head, body, color` etc. in a visual game system.*

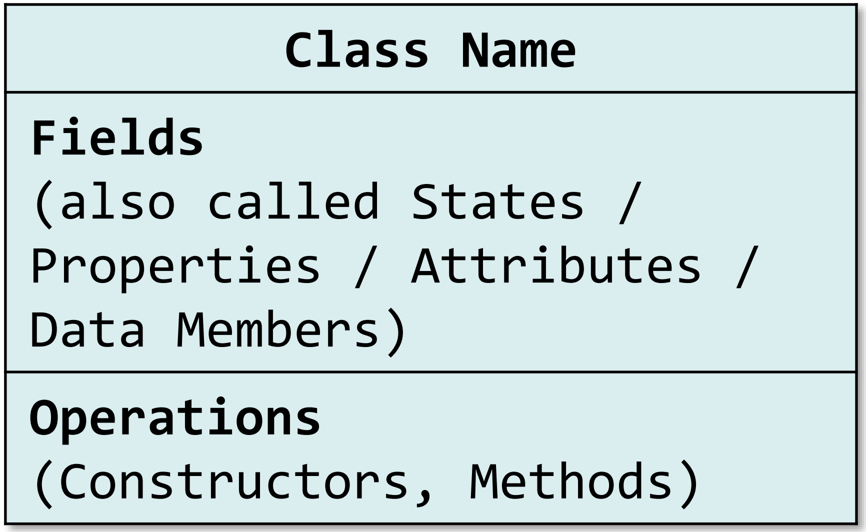
- A Student object may set a grade (say, via a ***method*** `setGrade()`), which requires a valid grade value input parameter but return no value

- A Student should be newly created / instantiated with ***constructor*** of 2 input parameters for setting its fields `sID`, `sName`, etc.

UML Class Diagram Representation

- A **class diagram** of the UML is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (constructors and methods), and the relationships among the classes

- Classes are represented with boxes that contain three compartments:
  - Class name,
  - Fields (Attributes/States), and
  - Operations / Behaviours
    - Constructors and Methods



Class Name

Fields
(also called States / Properties / Attributes / Data Members)

Operations
(Constructors, Methods)

Basic Notation of UML Class Diagram

- Field / Attribute notation:

```
<modifier><fieldname>:<fieldType>
```

- Constructor notation:

```
<modifier><ClassName>(<parameterName(s):parameterType(s)>)
```

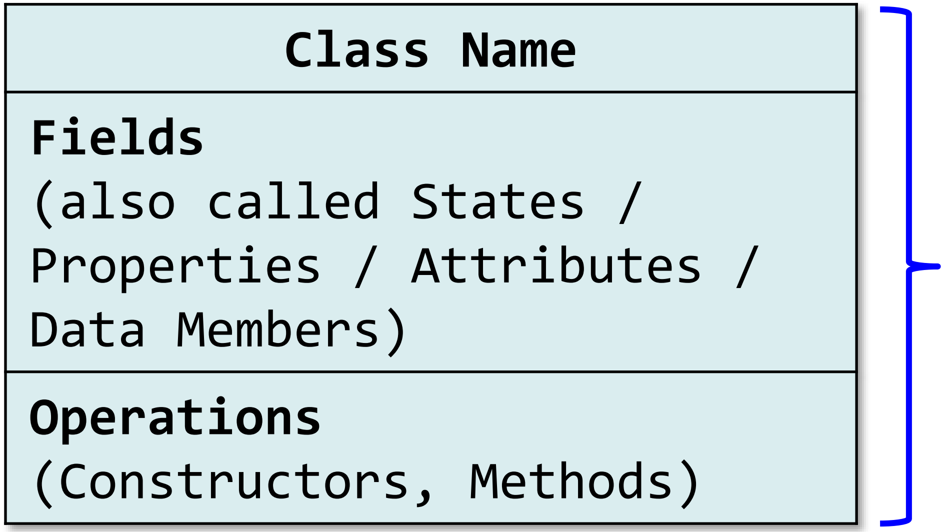
- Method notation:

```
<modifier><methodName>(<parameterName(s):parameterType(s)>):<returnType>
```

Visibility / Access Modifiers:

+	public
#	protected
-	private
	package

** In Java, "package" is the default visibility type, without any access modifier.*



Class Name

Fields
(also called States / Properties / Attributes / Data Members)

Operations
(Constructors, Methods)

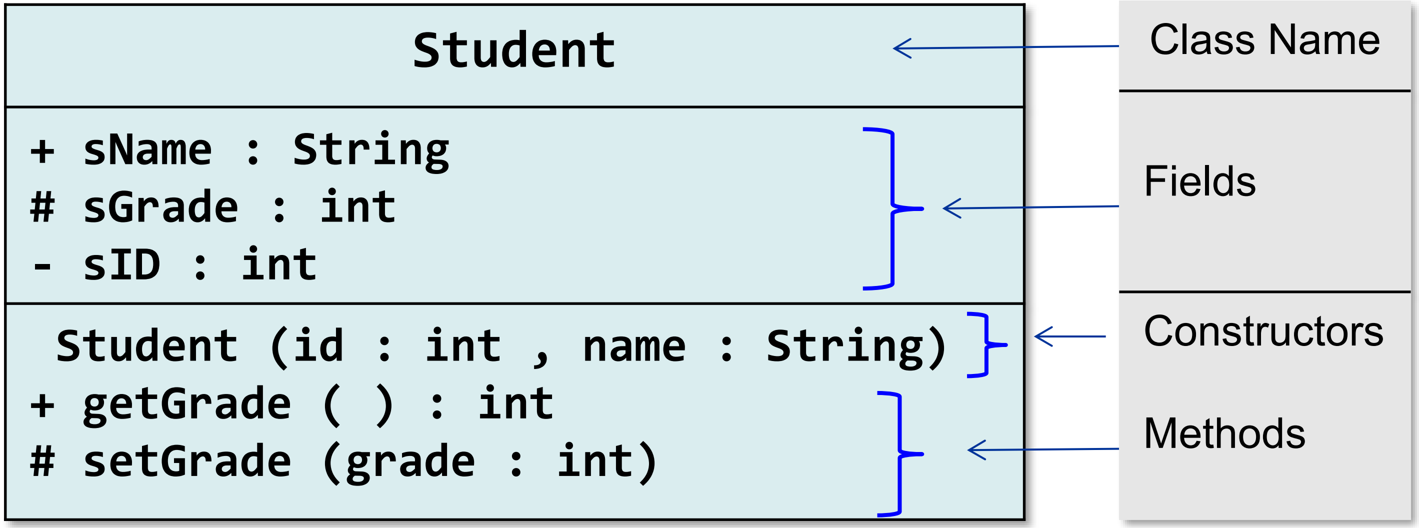
3 compartments

Example 1 – Student Class

Visibility / Access Modifiers:

+ `public`
# `protected`
- `private`
    `package`

** In Java, "package" is the default visibility type, without any access modifier.*



```
Student

+ sName : String
# sGrade : int
- sID : int

 Student (id : int , name : String)
+ getGrade ( ) : int
# setGrade (grade : int)
```

Class name:

> *Student*

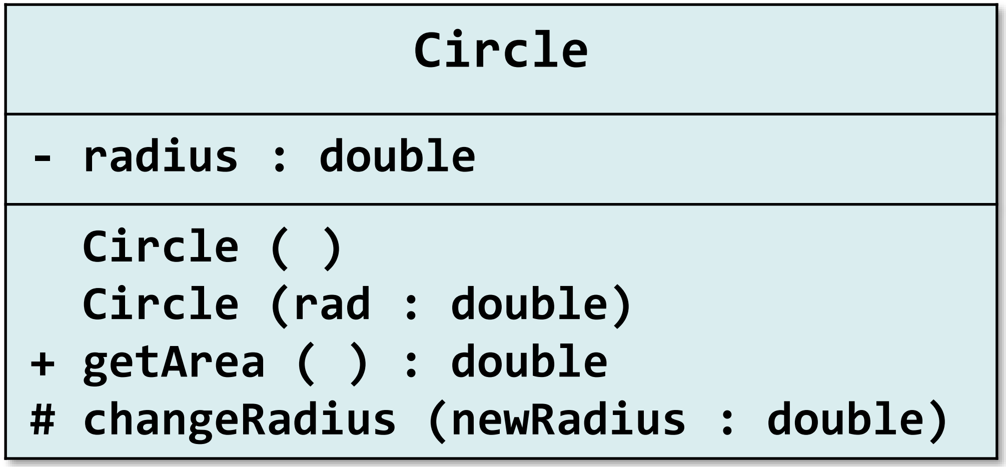
Number of public field:

> *1*

What is type of parameter for method `setGrade()` ?

> *int*

Example 2 – `Circle` Class



```
Circle
- radius : double
  Circle ( )
  Circle (rad : double)
+ getArea ( ) : double
# changeRadius (newRadius : double)
```

Class name:

> *Circle*

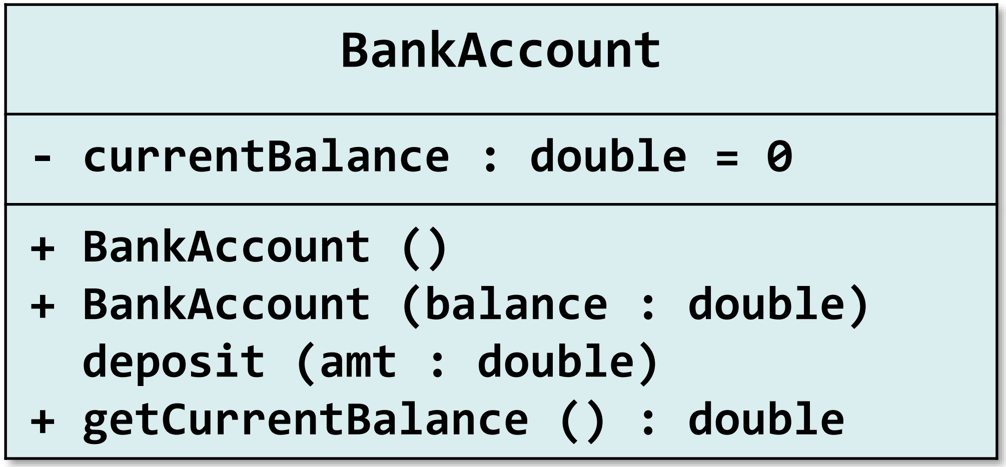
Number of private method:

> *0*

What is returned from method `changeRadius()` ?

> *nothing (void)*

Example 3 – BankAccount Class



```
BankAccount
- currentBalance : double = 0
+ BankAccount ()
+ BankAccount (balance : double)
  deposit (amt : double)
+ getCurrentBalance () : double
```

Class name:

> *BankAccount*

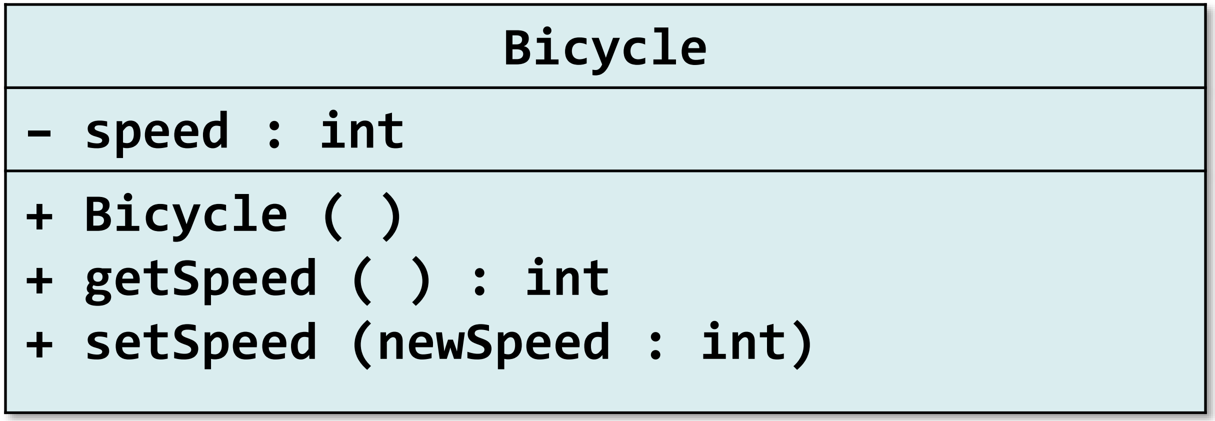
Number of Constructor (Having same name of class):

> *2*

What is returned from method `getCurrentBalance()` ?

> *double*

Example 4 – Bicycle Class



```
Bicycle
- speed : int
+ Bicycle ( )
+ getSpeed ( ) : int
+ setSpeed (newSpeed : int)
```

Number of Constructor (Having same name of class):

> *1*

What is the field name?

> *speed*

Defining Java Classes

- Learn how to define and code our own Java classes (e.g. based on our UML design) is the first step towards mastering the skills necessary in building large programs

- Classes we define ourselves are referred to as **programmer-defined (or user-defined) classes**

- It is in particular common to develop a Java program including a main class and other supporting classes as the coming examples

Implement a Java Class: Example Class `Bicycle`

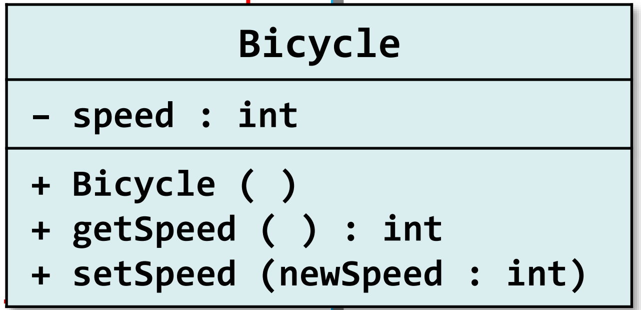
- Given a designed class model (e.g. with UML class diagram), we may implement / code the class in Java language accordingly. E.g.

```java
// Bicycle.java, a class modelling Bicycle
public class Bicycle { // class Bicycle declared here
   // Field (instance variable)
   private int speed;

   //Constructor: Initializes the field
   public Bicycle( ) {
        speed = 10;
   }

   //Method, get/return the speed of bicycle object
   public int getSpeed( ) {
        return speed;
   }

   //Method, set/assign the speed of bicycle
   public void setSpeed(int newSpeed) {
        speed = newSpeed;
   }
}
```

Class Declaration

Class Body, may include:

- **Fields,**
- **Constructors,**
- **Methods**



```
Bicycle
- speed : int
+ Bicycle ( )
+ getSpeed ( ) : int
+ setSpeed (newSpeed : int)
```

Syntaxes for Class Declaration

1) Class in a ***basic form***:

```
<modifier(s)> class <class name> {  <class body>  }
```

2) Class ***inherits from a specified (direct) superclass***:

```
<modifier(s)> class <class name> extends <superclass name> {
      <class body>
}
```

- Modifiers `<modifier(s)>` (e.g. `public`, `private` which determine what other classes can access)
- Class body `<class body>` is surrounded by a brace pair `{}`
- Class name `<class name>`, with initial letter capitalized by convention, preceded by the keyword `class`
- Name of the class's parent ("direct" superclass) `<superclass name>`, if any, preceded by the keyword `extends`
  - If not explicitly designate the "direct" superclass with the `extends` clause, the class's superclass is class `Object` (the root) in Java as in the first Basic Form

Syntaxes for Class Declaration

3) Class ***implements interface(s)***:

```
 <modifier(s)> class <class name> implements <interface(s)> {
      <class body>    }
```

4) Class ***inherits from specified superclass and implements interface(s)***:

```
<modifier(s)> class <class name> extends <superclass name>
      implements <interface(s)> {
      <class body>
}
```

- A comma-separated list of interfaces `<interface(s)>` implemented by the class, if any, preceded by the keyword `implements`, a class can *implement* more than one interface
- E.g.: Below declares a class `SubC`, which is the subclass of class `SuperC` (its direct superclass) and also implements two interfaces (`InA`, `InB`)

```
public class SubC extends SuperC implements InA, InB { //...
```

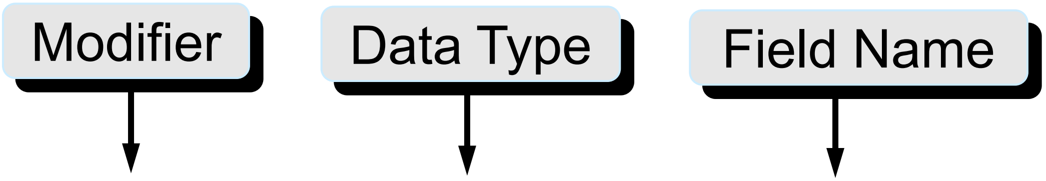
** Details of the topic will be further discussed in later unit*

Field (Attributes or Data Member)

- **Fields** (for the states / attributes / data members of a class) are declared in a class, outside all methods or constructors

```
<modifier(s)> <data type> <field name> ;
```

Examples:



Modifier

Data Type

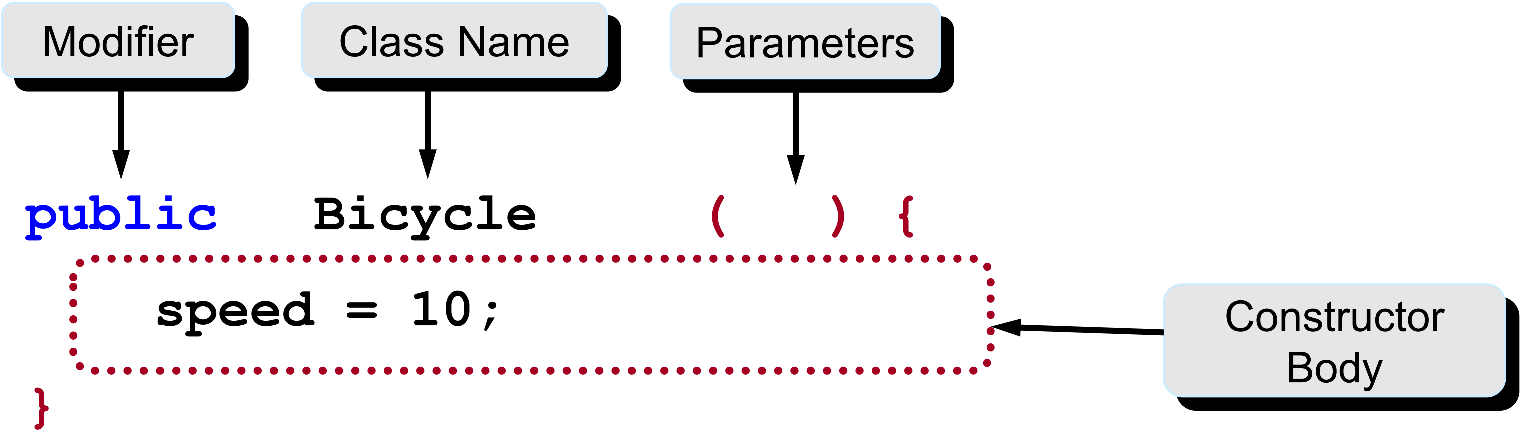
Field Name

```
private    String    ownerName ;

private    int       speed ;
```

Constructor

- A **constructor** acts like a special kind of method that is called when an object is instantiated (newly created) with keyword **new**.

```
<modifier> <class name> (<parameter(s)>){
    <statement(s)>
}
```



Modifier

Class Name

Parameters

Constructor Body

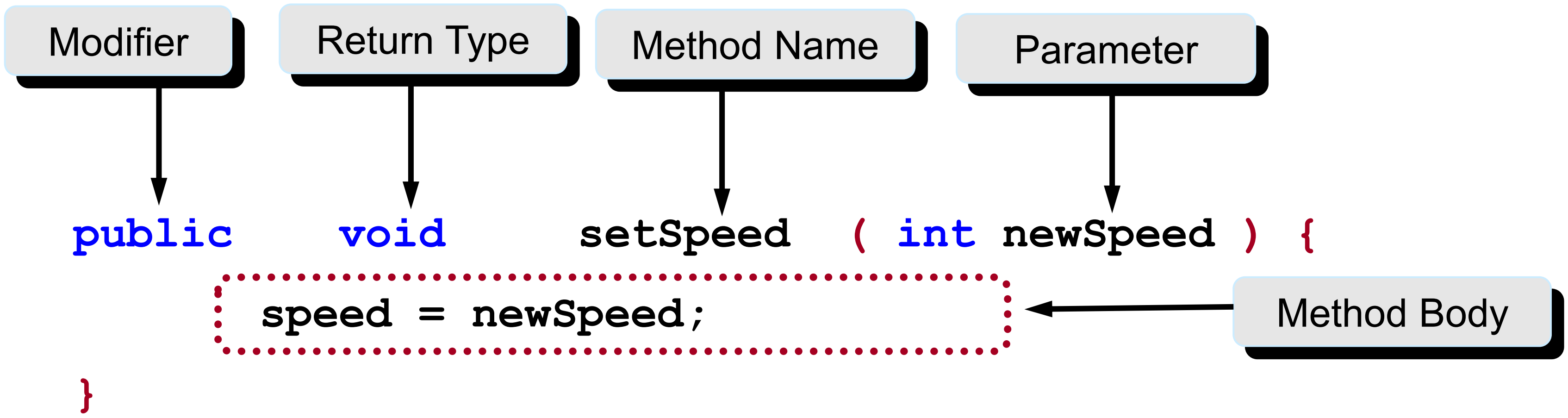
```
public  Bicycle    (   ) {
    speed = 10;
}
```

Method

- **Method** defines “behavior” or activity of a class, which essentially is program “module” containing statements to perform specific tasks

```
<modifier(s)> <return type> <method name> (<parameter(s)>){
       <method body>
}
```

Example:



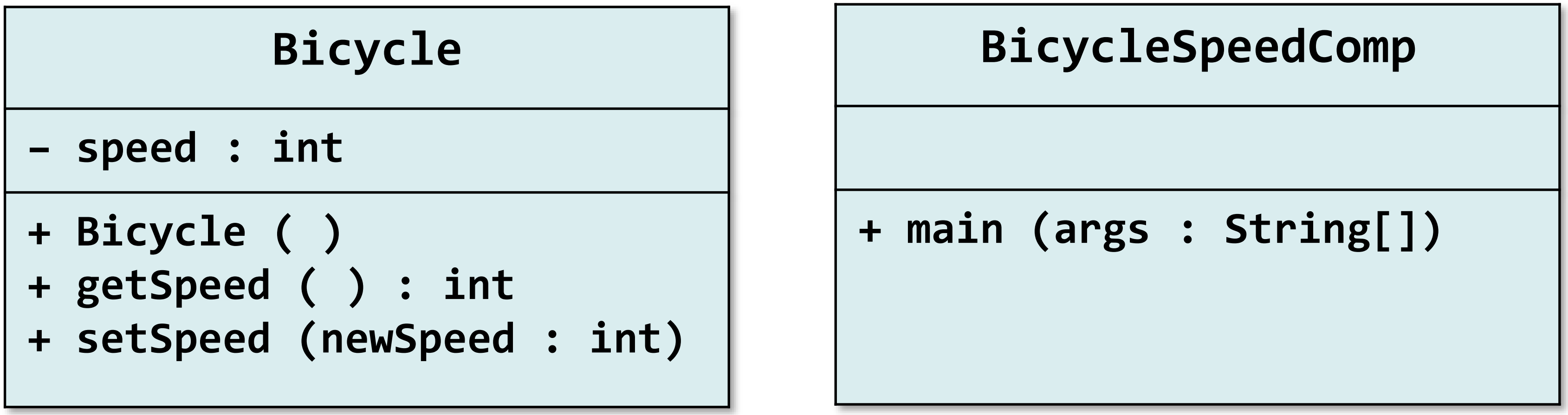
```
public    void    setSpeed  ( int newSpeed ) {
       speed = newSpeed;
 }
```

Main Class: `BicycleSpeedComp` Use `Bicycle` Class

- A class having **`main()` method** (where program starts) often initializes / sets up the system, and is called **“main” class of the program**
  - `BicycleSpeedComp` is the main class in this case, using class `Bicycle`.

```
// BicycleSpeedComp.java, the main class to start the program
public class BicycleSpeedComp {
    public static void main(String[] args) {
        Bicycle bike1, bike2;
        int  speed1, speed2;
        bike1 = new Bicycle( );
        bike1.setSpeed(35); // (method calling) outside its class
        //Create and assign values to bike2
        bike2 = new Bicycle( );
        //Output the information
        speed1 = bike1.getSpeed( );
        speed2 = bike2.getSpeed( );
        if (speed1 > speed2)
            System.out.println("Bicycle 1 is faster than Bicycle 2.");
    }
}
```

UML Class Diagram
(Bicycle and BicycleSpeedComp)



```
Bicycle
- speed : int
+ Bicycle ( )
+ getSpeed ( ) : int
+ setSpeed (newSpeed : int)
```

```
BicycleSpeedComp
+ main (args : String[])
```

COMPILE ALL Java Files

```
C:\>javac *.java
```

RUN, on console

```
C:\>java BicycleSpeedComp
```

The Program Structure and Source Files

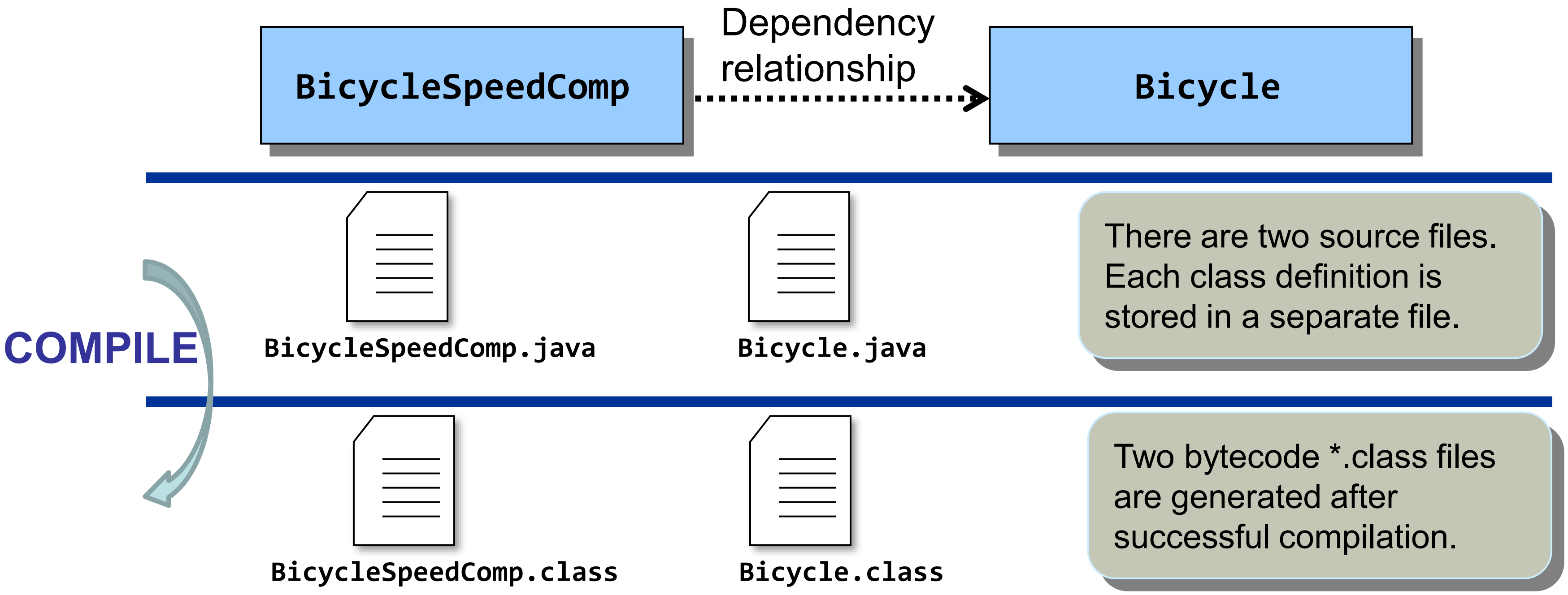
*To **Compile** the program:*

1. `javac Bicycle.java` (compile)
2. `javac BicycleSpeedComp.java` (compile)

*To **Run** the program:*

3. `java BicycleSpeedComp` (run)

** Remark: Step 1 is optional in this case. We may also use `javac *.java` approach*



There are two source files. Each class definition is stored in a separate file.

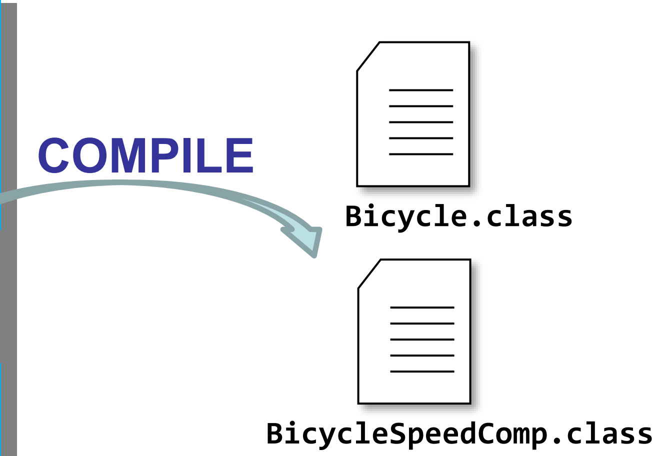
Two bytecode *.class files are generated after successful compilation.

Multiple Classes in One Java File

- We may define multiple classes in a single source file, *but not recommended in our course*.
  - If there is any, there could *ONLY be one* `public class`, which must have the same name as the source Java file name.
    - For example, we can define `public class BicycleSpeedComp` in the file `BicycleSpeedComp.java`, and also define another non-public `class Bicycle` in the same `BicycleSpeedComp.java` file.

```
// File BicycleSpeedComp.java
// Version of one file having two class
public class BicycleSpeedComp {
    public static void main(String[] args) {
// ... And more
        Bicycle bike1, bike2;
    // ... And more
    }
}

class Bicycle { // class Bicycle declared
// ... And more
    public Bicycle( ) {
        speed = 10;
    }
// ... And more
}
```

One source file contains 2 class.

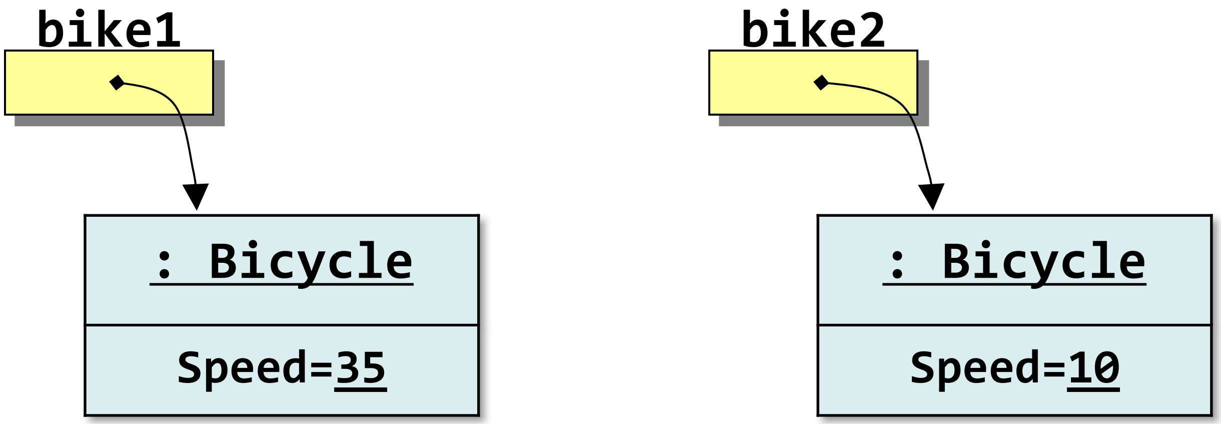


Two bytecode `*.class` files are generated, after successful compilation.

Multiple Instances

- Once the `Bicycle` class is defined, we can create multiple objects/instances of this `Bicycle` class
- Similar to other classes such as `String` class, each unique object created with **new** operator has its own field (such as `speed` of Bicycle in the case below).

```
Bicycle bike1, bike2;

bike1 = new Bicycle( );
bike1.setSpeed(35);

bike2 = new Bicycle( );
```

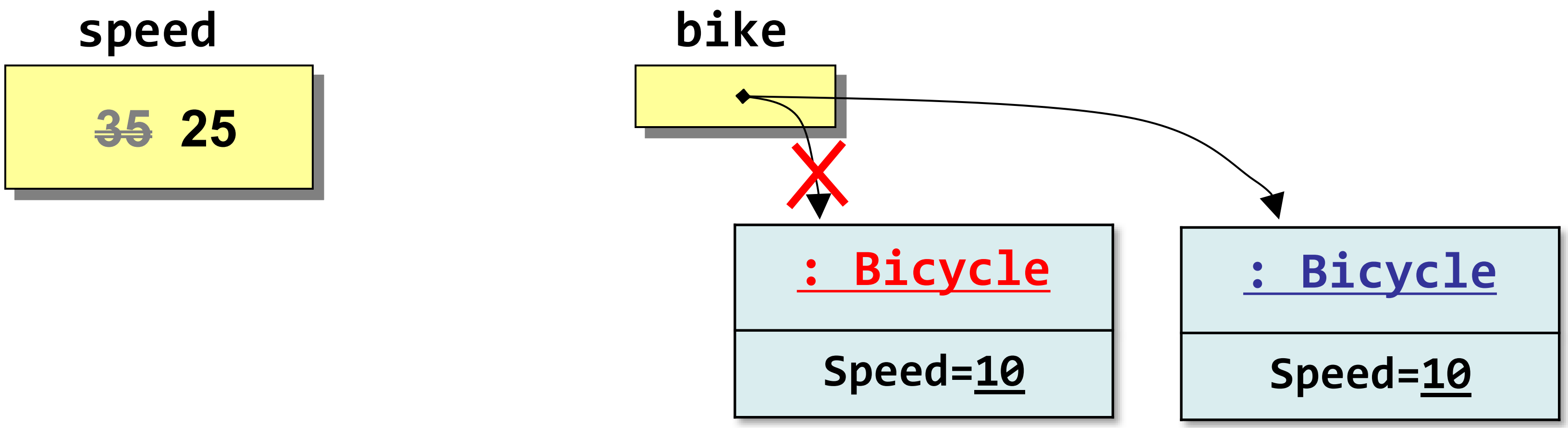


Re-Assignment in Primitive Type vs. Reference Type

- For *primitive* type such as `int`, re-assigning a value to the variable will update the content (the actual new value)
- For *reference* type such as class `Bicycle`, re-assigning a value to the variable will update the content (the reference to a new object)

```
// Primitive type
int speed;
speed = 35;
speed = 25;
```

```
// Reference type
Bicycle bike;
bike = new Bicycle();
bike = new Bicycle();
```



speed

~~35~~ 25

bike

: Bicycle

Speed=10

: Bicycle

Speed=10

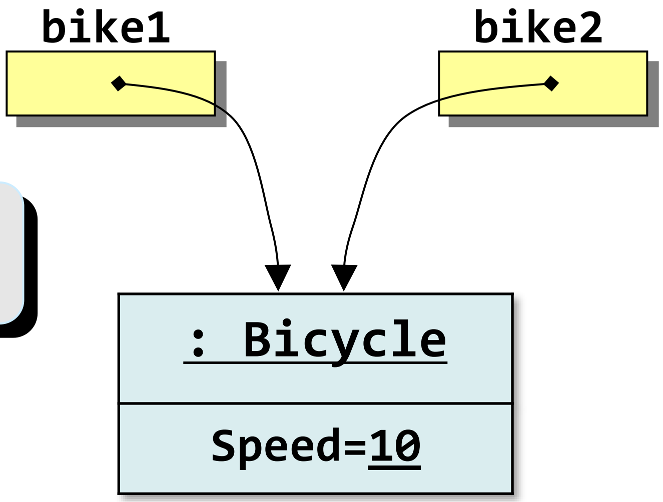
Two References to a Single Object

```
Bicycle bike1, bike2;     // 1
bike1 = new Bicycle();    // 2
bike2 = bike1;            // 3
```

1. Variables are allocated in memory, for object reference

2. The reference of the newly created object is assigned to **bike1**

3. The reference in **bike1** is assigned to **bike2**



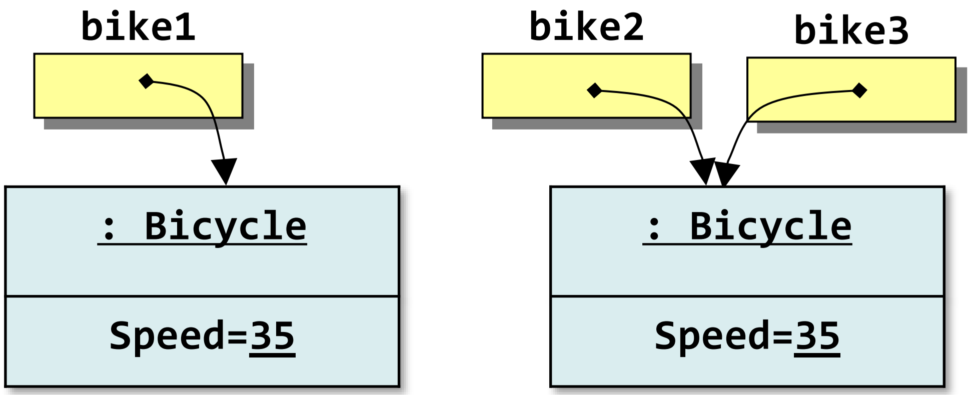
Using == Operator to Compare Objects

- With reference types (e.g. class or array), “equal to” operator **==** checks if they have same *reference (Similar to the case of `String` with `new` operator)*

```
Bicycle bike1, bike2, bike3;
bike1 = new Bicycle( ); bike1.setSpeed(35);
bike2 = new Bicycle( ); bike2.setSpeed(35);
bike3 = bike2;
if (bike1 == bike2) // == "equal to" operator, compares reference
        System.out.println("They do refer same object");
else    System.out.println("They do NOT refer same object");
if (bike2 == bike3) // == "equal to" operator, compares reference
        System.out.println("They do refer same object");
else    System.out.println("They do NOT refer same object");
```

false

true



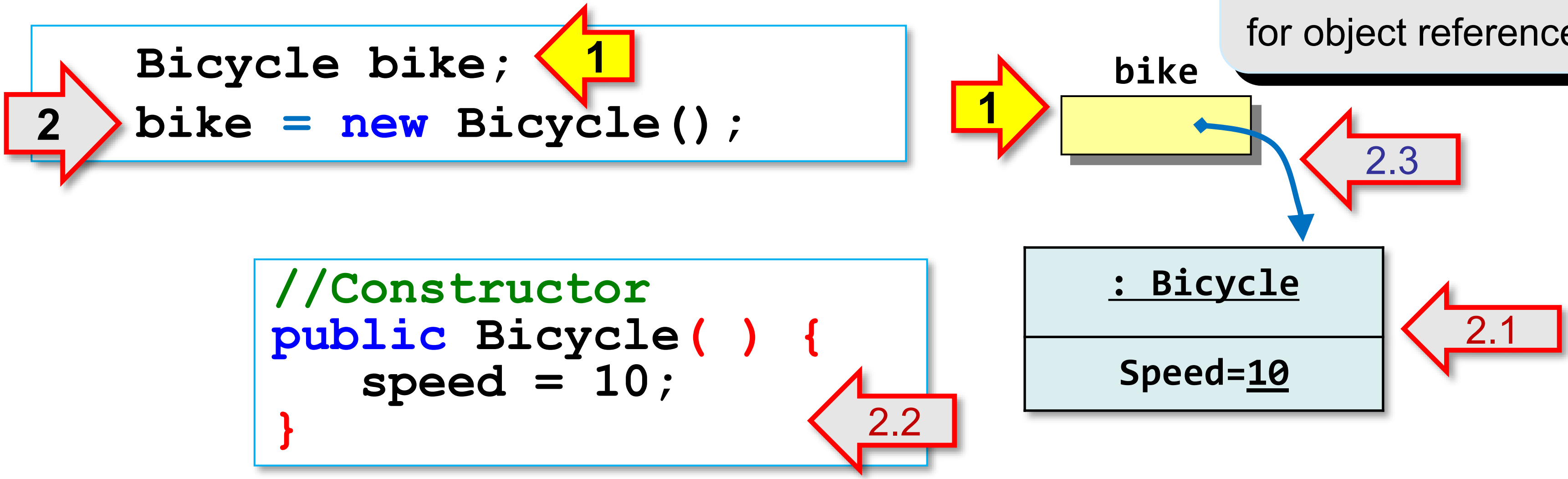
Constructor - Initializing Objects

- A ***constructor*** acts like a special kind of method that is called when an object is instantiated (newly created)
  - The body of the constructor will be executed if the constructor is invoked / called.
  - Constructor must have the same name as the class itself
  - Constructor does **not** have a return type (not even **`void`**)
  - Constructor cannot be called / invoked directly, it is invoked *automatically* when you create a new object using the **`new`** operator
- It plays the main role for *initializing objects* (e.g. set initial values for fields, etc.)

How Does Constructor Work?

- To create a new **`Bicycle`** object called **`bike`**, a constructor is called by the **`new`** operator
  - `new Bicycle()` creates space in memory for the object, and initializes its fields in the constructor

1. Variable **`bike`** is allocated in memory, for object reference



```
Bicycle bike;
bike = new Bicycle();
```

```
//Constructor
public Bicycle( ) {
    speed = 10;
}
```

2. The reference of a newly created **`Bicycle`** object is assigned to variable **`bike`**

- 2.1. Memory is created for a new `Bicycle` object
- 2.2. Related constructor is then called and executed
- 2.3. Object is assigned to and referenced by the variable `bike`.

Default Constructor

- You should define your own constructor(s)
  - It is allowed to define more than one constructor (overloaded constructors, *to be discussed in later unit*)

- *If no constructor is defined* for your class, a ***default constructor*** is automatically generated by compiler
  - Default constructor is a *no-argument constructor* which does nothing, *e.g.*

```
public class Person { } // without define a constructor
```

Similar to:

```
public class Person {
   public Person( ){ } // default constructor added
}
```

- Good practice: define our own constructor(s)

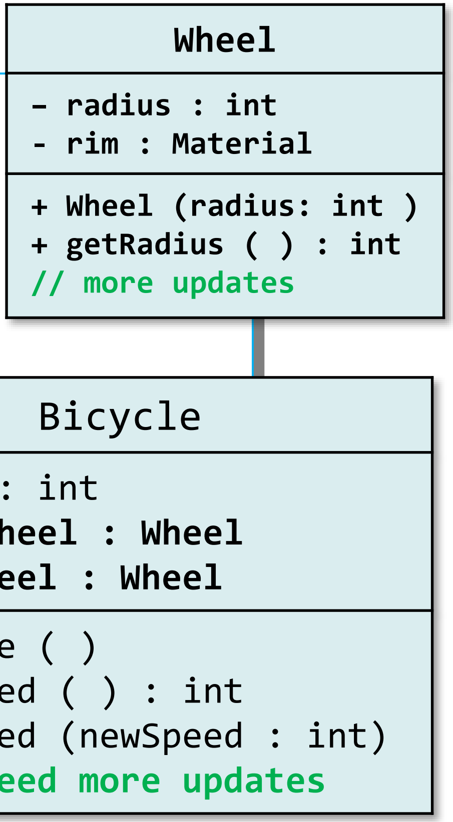
** In fact, calling its superclass's constructor should be done first in a subclass constructor body, to be discussed in later unit*

More Complicated Java Classes

- It is common to model a complicated entity with more than one class
  - E.g. our class `Bicycle` may be updated to include two new fields (`frontWheel` , `rearWheel`) of another class type `Wheel` as below

    * There should be more related updates, e.g. more proper methods

```
// Bicycle.java, a class modelling Bicycle
public class Bicycle { // class Bicycle declared here

    // Fields (instance variable)
    private int speed;
    Wheel frontWheel; // a field of another class, Wheel
    Wheel rearWheel;

    //Constructor: Initializes the field
    public Bicycle( ) {
        speed = 10;
    }

// ..
// MORE UPDATES: Fields, Constructors, Methods


}
```



```
Wheel
- radius : int
- rim : Material
+ Wheel (radius: int )
+ getRadius ( ) : int
// more updates
```

```
Bicycle
- speed : int
- frontWheel : Wheel
- rearWheel : Wheel
+ Bicycle ( )
+ getSpeed ( ) : int
+ setSpeed (newSpeed : int)
// may need more updates
```

More Complicated Java Classes

- Suppose our earlier main class (`BicycleAccountTest`) is updated to also include another class **`Account`** below

```
public class Account {

    private String ownerName;

    private double balance;

    public Account( ) {
        ownerName = "Unassigned";
        balance = 0.0;
    }

    public void add(double amt) {
        balance = balance + amt;
    }

    public void deduct(double amt) {
        balance = balance - amt;
    }

    public double getCurrentBalance( ) {
        return balance;
    }
  //...                          // Page 1
```

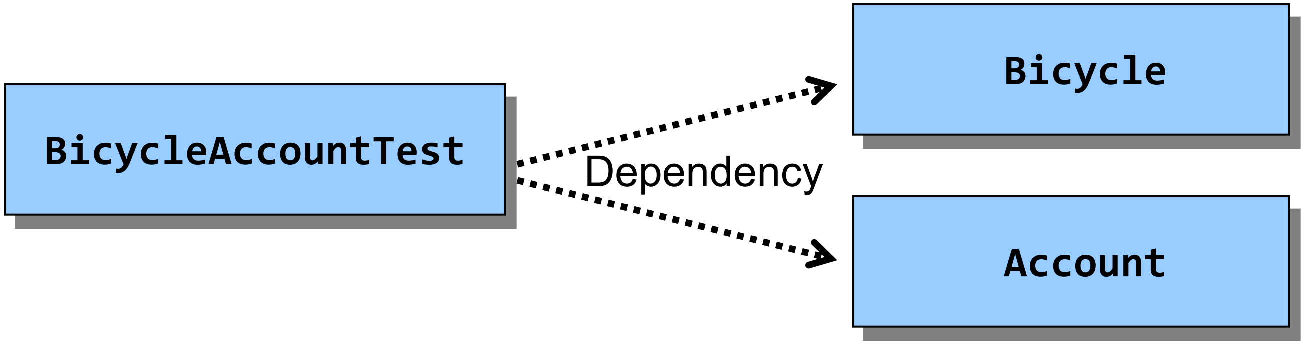
```
    public String getOwnerName( ) {

        return ownerName;
    }

    public void setInitialBalance
                       (double bal) {

        balance = bal;
    }

    public void setOwnerName
                      (String nStr) {

        ownerName = nStr;
    }

}

                                // Page 2
```

More Complicated Java Classes

- Updated example program involves more classes defined:
  - `Bicycle`, `Account`, and the main class `BicycleAccountTest` below

```
class BicycleAccountTest {

    //This sample program uses both the Bicycle and Account classes

    public static void main(String[] args) {

         Bicycle bike;
         Account acct;

         String  myName = "Jon Java";

         bike = new Bicycle( );

         acct = new Account( );
         acct.setOwnerName(myName);
         acct.setInitialBalance(250.00);

         acct.add(25.00);
         acct.deduct(50);

         //Output some information
         System.out.println (acct.getOwnerName());
         System.out.print (" rides at speed = " + bike.getSpeed() + " and");
         System.out.println(" has $" + acct.getCurrentBalance());
    }
}
```

The Program Structure



`BicycleAccountTest.java` `Bicycle.java` `Account.java`

To run the program:

~~1. `javac Bicycle.java` (compile)~~
~~2. `javac Account.java` (compile)~~
3. `javac BicycleAccountTest.java` (compile)
4. `java BicycleAccountTest` (run)

COMPILE ALL Java Files

```
C:\>javac *.java
```

RUN, on console

```
C:\>java BicycleAccountTest
```

Note: In this case, we only need to compile the class once, and recompile only when we made changes in the code. We may use `javac *.java` approach to compile ALL Java source files.

Class Fields/Methods vs. Instance Fields/Methods

- So far, each instance / object of a class has its own set of fields and methods

- However, it is also common that we need to define some specific fields and methods for the whole class:
  - They do not belong to any specific individual instance
  - They belong to the whole class or all instances

- We call them **class fields** and **class methods** (or *static* fields and *static* methods, and we use keyword `static` in Java to represent them).

Fields: Instance vs. Class

- **Instance field** (also called *non-static field*, *instance variable*, or just simply *field*)
  - For maintaining information *specific* to individual instances, e.g.
    - Field `speed` of class `Bicycle`: each `Bicycle` object maintains its own `speed` (instance), and different `Bicycle` objects have different speed
    - Field `balance` of class `BankAccount`: each `BankAccount` object maintains its own `balance` (instance) field representing the amount of its own current balance

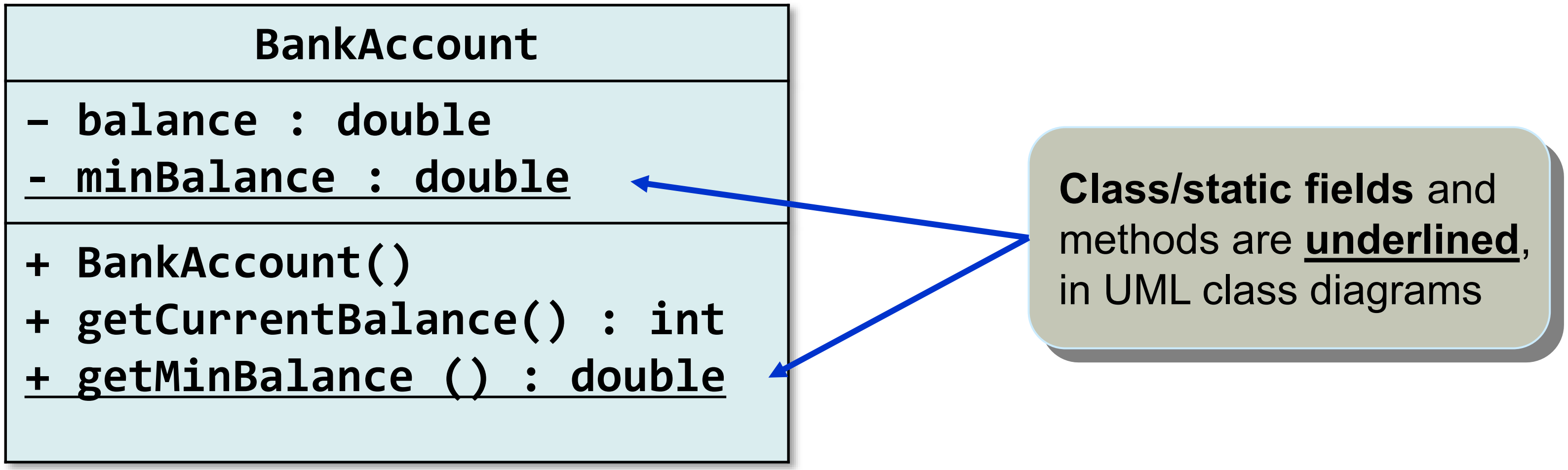
- **Class field** (also called *static field* or *class variable*)
  - For maintaining information often obtained from not only one specific instances, or shared by all instances
    - Field `avgSpeed` of class `Bicycle`: class `Bicycle` maintains only one `avgSpeed` (class) field representing the average speed obtained from all instances of the whole class
    - Field `minBalance` of class `BankAccount`: class `BankAccount` maintains only one `minBalance` (class) field representing the minimum balance among all instances of the class

Methods: Instance vs. Class

- **Instance Method** (also called *non-static method*, or just simply *method*)
  - A method defined *specific* for an instance or object, often working with its own instance fields and other instance methods, e.g.
    - Method `getCurrentBalance()` of class `BankAccount`: each `BankAccount` object has its "own" `getCurrentBalance()` (instance) method, which returns the amount of its "own" balance (instance field)
- **Class Method** (also called *static method*)
  - A method defined for the whole class, which cannot refer to any specific instance (non-static) fields/methods because values for those only exist for instances of the class, e.g.
    - Method `getMinBalance()` of class `BankAccount`: this (class) method `getMinBalance()` is not working for any specific instance and its fields, but for the whole class to obtain the class field `minBalance`

Class Fields and Methods

- ***UML Class diagram***: A Simplified Example



BankAccount

- balance : double
- minBalance : double

+ BankAccount()
+ getCurrentBalance() : int
+ getMinBalance () : double

Class/static fields and methods are **underlined**, in UML class diagrams

- ***Define a class field / method*** in a similar way as instance field / method, with the keyword **static**

```
public class BankAccount {
  private double balance;
  private static double minBalance; // class field
  public double getBalance(){ return balance; }
  public static double getMinBalance(){ // class method
    return minBalance;
  }
// ... More codes
}
```

Access Class Fields / Methods, with Class Name

- ***Access a class field / method*** directly with the class name, without creating an instance, with general syntax:

```
    <ClassName>.<ClassField>
    <ClassName>.<ClassMethod>
```

- Example codes of comparing ways to access instance field/method (with the specific object name) and to access class field/method (with the class name)

```
BankAccount peterAcc = new BankAccount();
// ..
double peterBal = peterAcc.getBalance(); // call a field method
// ..
double minBal = BankAccount.getMinBalance(); // call a class method

// Other Example Java Standard Classes: Math & Color
Math.PI                 // public class constant field
Color.GREEN             // public class constant field
Math.sqrt(12.3)         // public class method
Math.random()           // public class method
```

Example Usage of Class Field

- It is most common to define a series of *public class constants* for external use, with keywords **`static`** and **`final`**

- A class field is comparatively less common, especially for external use. Below shows only an example of using class field to keep track on the total number of `Bicycle` objects created (via constructor calling)

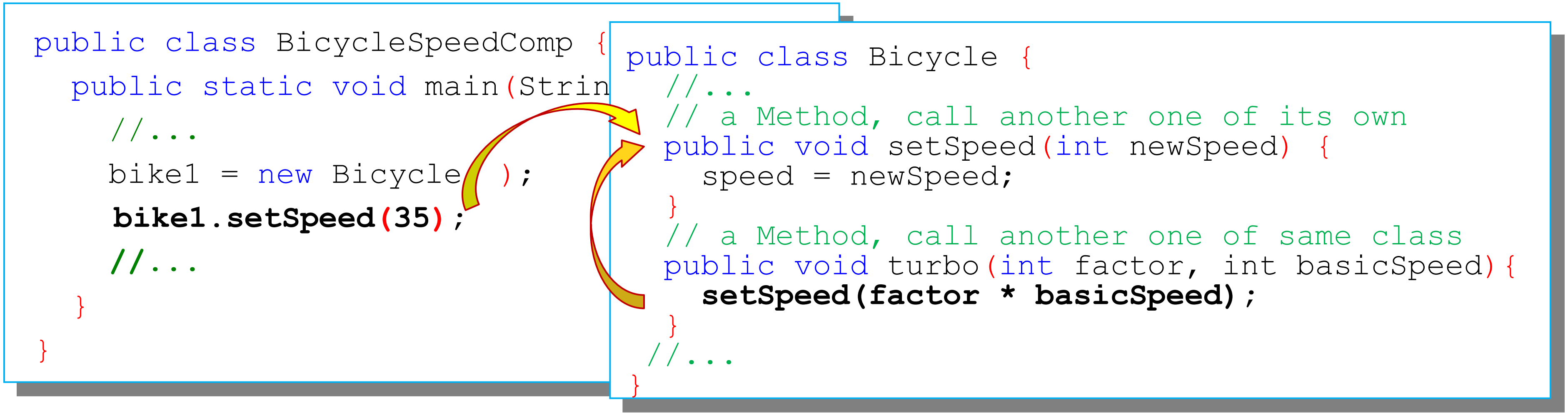
```
public class Bicycle {
  public static final int TURBO_SPEED = 90; // public class constant
  public static final int SLOW_SPEED = 2; // public class constant
  private int speed;
  private static int numberOfBicycles = 0; // class field

  public Bicycle(int startSpeed) {
    speed = startSpeed;
    // increment total number of Bicycles
    numberOfBicycle++;
  }
}
```

Method Calling

- So far, we have been calling a method of another class / object, with dot-notation with syntax **`<objectName>.<method>`**, e.g.

```
bike1 = new Bicycle( );
bike1.setSpeed(35);
```

- It is also common to call a method of a class from another method of the same class (`Bicycle` below)
  - In this case, we often refer to another method **without** dot notation. This also applies to accessing a field within the class.
  - E.g. the modified class `Bicycle` below adds a new method `turbo()` which calls another method `setSpeed()` directly, without dot notation.

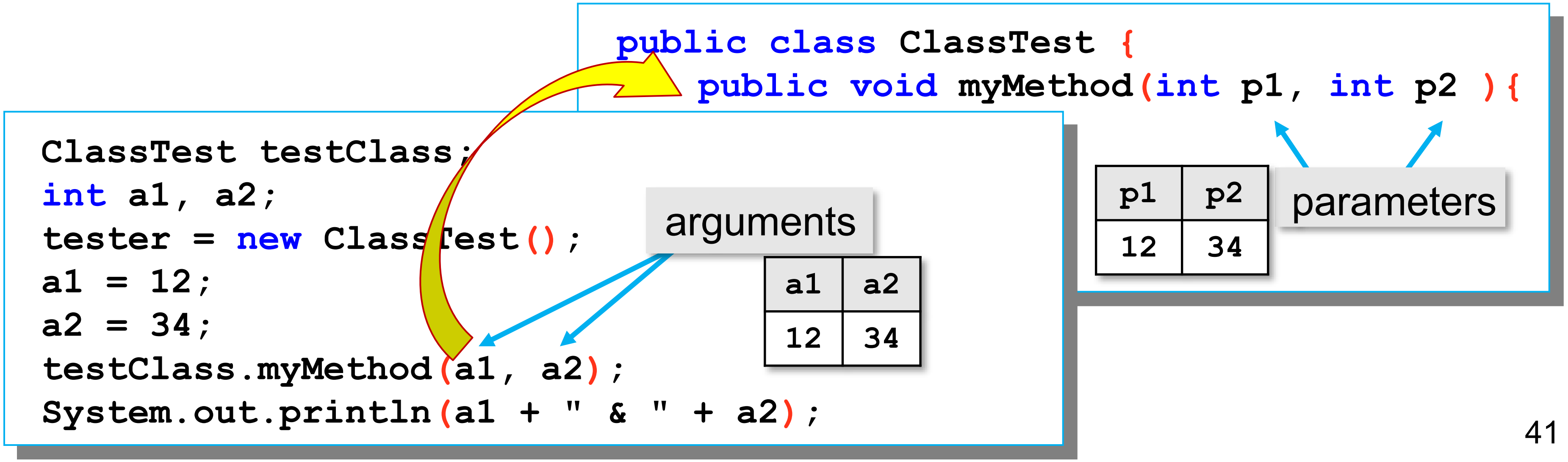


```
public class BicycleSpeedComp {
  public static void main(Strin
    //...
    bike1 = new Bicycle( );
    bike1.setSpeed(35);
    //...
  }
}
```

```
public class Bicycle {
  //...
  // a Method, call another one of its own
  public void setSpeed(int newSpeed) {
    speed = newSpeed;
  }
  // a Method, call another one of same class
  public void turbo(int factor, int basicSpeed){
    setSpeed(factor * basicSpeed);
  }
 //...
}
```

Arguments vs. Parameters

- An **argument** is a value we pass to a method *in method calling*.
  - It is the actual value that is passed in when the method is invoked.
- A **parameter** is a placeholder in the called method to hold the value of the passed argument
  - It refers to the list of variables *in a method declaration*
- In method calling, value of argument is passed to that of parameter
- Example:



```
public class ClassTest {
  public void myMethod(int p1, int p2 ){
```

p1	p2
12	34

parameters

```
ClassTest testClass;
int a1, a2;
tester = new ClassTest();
a1 = 12;
a2 = 34;
testClass.myMethod(a1, a2);
System.out.println(a1 + " & " + a2);
```

arguments

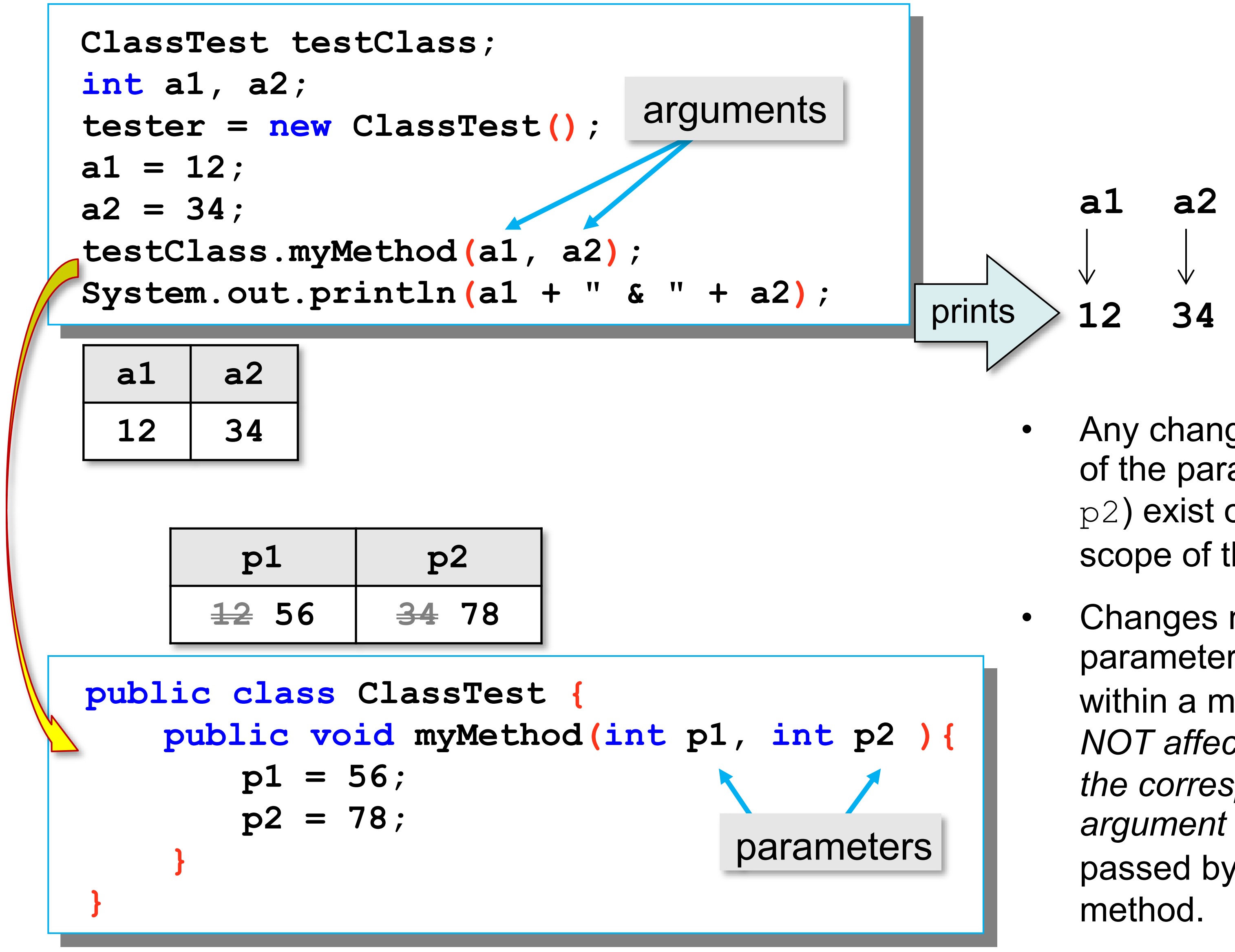
a1	a2
12	34

Pass-by-Value Parameter Passing

- When a method is called, *value of the argument is passed* to parameter (not the argument itself)
  - Separate memory space (for the parameter) is allocated to store this passed value in the method
  - This way of passing the value of arguments is called a ***pass-by-value*** or *call-by-value scheme*

- Since separate memory space is allocated for each parameter during the execution of the method:
  - The parameter is local to the method, and any changes to the values of the parameters exist only within the scope of the method
  - Changes made to the parameter within a method body will *NOT affect the value of the corresponding argument* passed by the calling method

Matching Arguments and Parameters



```
ClassTest testClass;
int a1, a2;
tester = new ClassTest();
a1 = 12;
a2 = 34;
testClass.myMethod(a1, a2);
System.out.println(a1 + " & " + a2);
```

a1	a2
12	34

p1	p2
~~12~~ 56	~~34~~ 78

```
public class ClassTest {
    public void myMethod(int p1, int p2 ){
        p1 = 56;
        p2 = 78;
    }
}
```

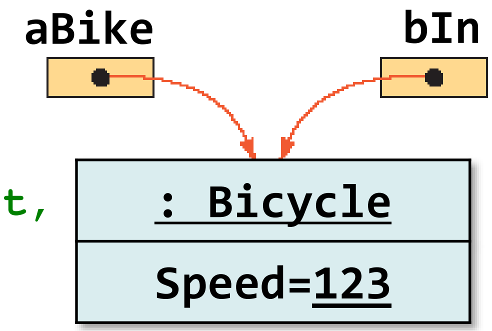
- Any changes to the values of the parameters (`p1` and `p2`) exist only within the scope of the method.
- Changes made to the parameter (`p1` and `p2`) within a method body will *NOT affect the value of the corresponding argument* (`a1` and `a2`) passed by the calling method.

Passing *Reference* Data Type Arguments

- Parameters of reference type, such as objects of specific classes, are also passed *by "value"* into methods
- This means when method returns, *arguments remain unchanged (They still reference to same objects as before)*
- However, the referenced object (e.g. its field values) *could* be changed in the method if accessing properly

E.g.

```
aMethod(aBike); // argument aBike
// HERE aBike stills references the same object,
  // after the method call above
void aMethod(Bicycle bIn){ // parameter bIn
// HERE referenced object's fields could be changed, e.g. below
   bIn.setSpeed(123);
}
```



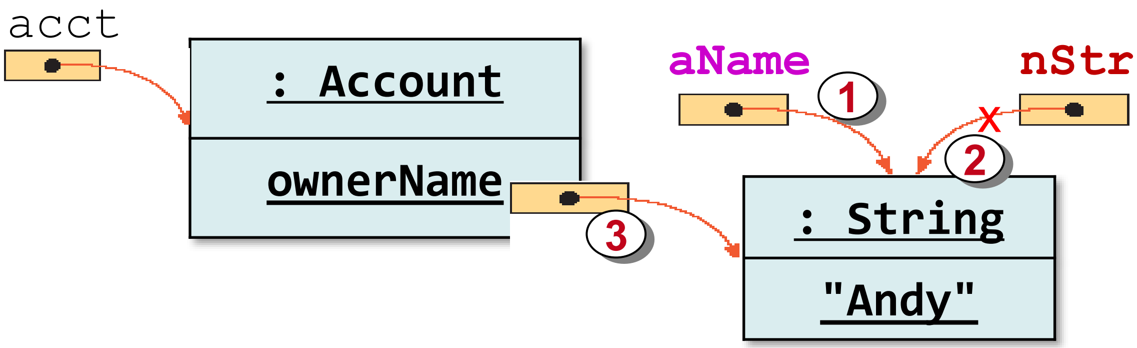
Passing Objects to a Method

- Passing an object will pass the reference of an object
  - It means NO duplicate of an object will be created in the called method
  - Example below: the same object (of `String`) containing `"ANDY"` are referenced by both argument `aName` and parameter `nStr`, at the moment of calling the method `setOwnerName()` of `Account acct`
  - Parameter `nStr` will be deleted after the method has finished.

* Parameters (and local variables) will be deleted after method has finished and returned.

```
// . . . Main Class
Account acct = new Account();
1 String  aName = new String("ANDY");
acct.setOwnerName(aName);
// . . .
```

```
public class Account {
   private String ownerName;
// . . .
   public void setOwnerName      2
                   (String nStr){
      ownerName = nStr;
   }  3
```



* will be deleted after method returns, for method parameter.

Parameter Passing: Key Points

1) Arguments are matched to the parameters from left to right. The *data type of an argument must be assignment-compatible* with the data type of the matching parameter.

2) Number of arguments in the *method call must match number of parameters* in the method definition

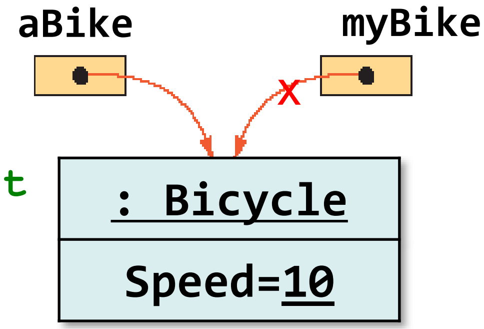
3) Parameters and arguments *do not have to have the same name*

4) Parameters are input to a method, and they are local to the method. *Changes made to the parameters will not affect the value of corresponding arguments*

** These issues of arguments and parameters are also applied to constructor calling*

Returning an Object from a Method

- Similar to passing the reference of an object in method calling, returning an object back to the caller is actually returning a reference (or an address) of an object
  - This means we are not returning a copy of an object, but only the reference of this object
  - Local variable `myBike` will be deleted after the method has finished.



E.g.

```
aBike = createBike(); // return & assign a object
// ...
// BELOW method will return an object of Bicycle
Bicycle createBike(){ // return type: Bicycle
// HERE referenced object's fields could be changed,
    Bicycle myBike = new Bicycle( );
    return myBike; // return myBike, an object of Bicycle
}
```

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:

  - Deitel, P. and Deitel H. (2017) "Java How To Program, Early Objects", 11ed, Pearson.
  - Liang, Y.D. (2017) "Introduction to Java Programming and Data Structures", Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) "An Introduction to Object-Oriented Programming with Java", 5ed, McGraw Hill.
  - Oracle Corporation, "Java Language and Virtual Machine Specifications" https://docs.oracle.com/javase/specs/
  - Oracle Corporation, "The Java Tutorials" https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/