innovate achieve lead

BITS Pilani

Pilani Campus

# Classes & Objects

# Agenda

- Defining Classes and Access Modifiers
- Creating Objects
- Role of Constructors
- Accessing Instance Fields and Methods
- Local Variables vs. Instance Variables
- UML representation of a Class
- Method Overloading
- Constructor Overloading
- Objects as Parameters to Methods

# A Class Is a Type

- A *class* is a special kind of programmer-defined type, and variables can be declared of a class type (recall structs from C)
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases
    - "X is of type A"
    - "X is an object of the class A"
    - "X is an instance of the class A"

    } mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

# The Contents of a Class Definition

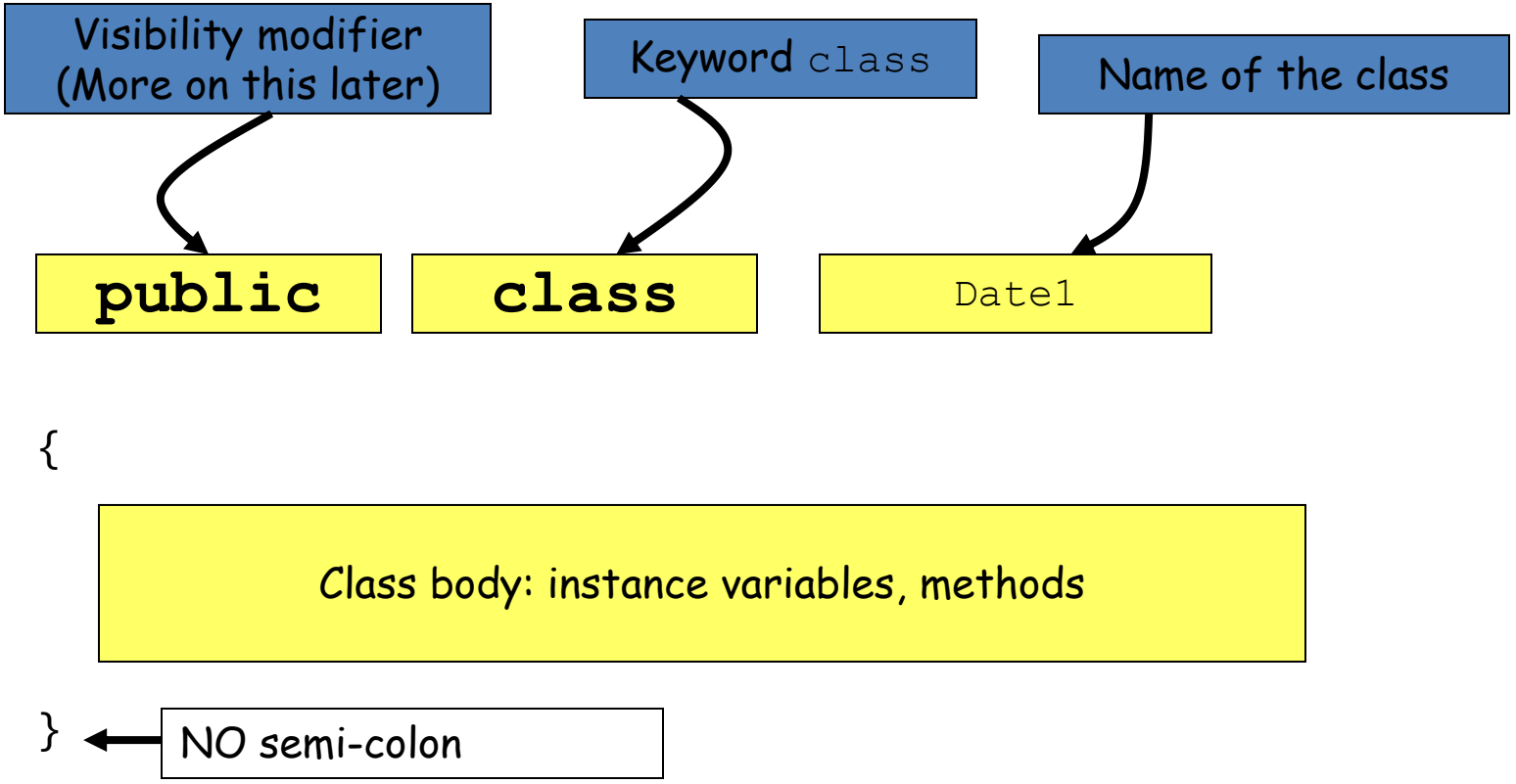
- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

# Primitive Type Values vs. Class Type Values

- A primitive type value (int, double, char) is a single piece of data

- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

# Anatomy of a Java Class

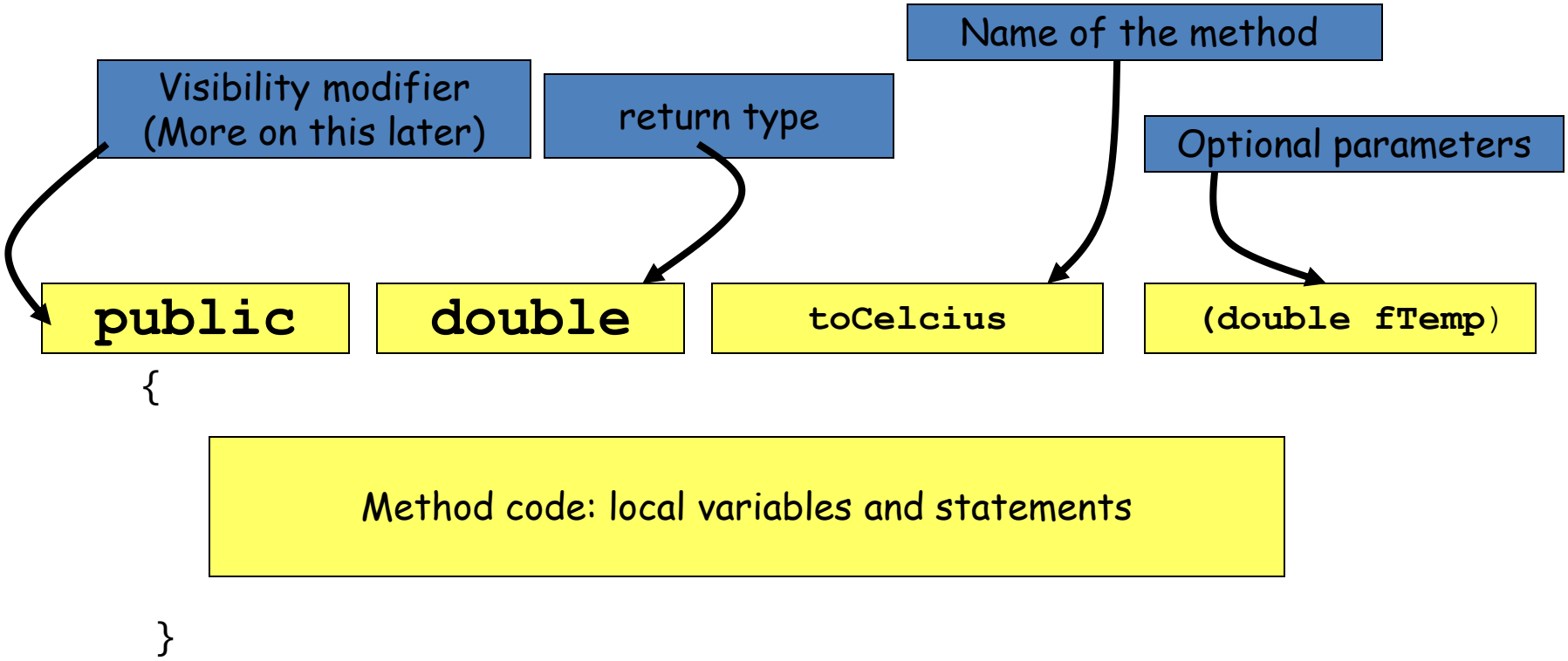


# Declaring Member Variables

- Types of variables
  - **Member variables in a class—these are called *fields***
  - **Variables in a method or block of code—these are called *local variables***
  - **Variables in method declarations—these are called *parameters***
- Field declarations are composed of three components, in order:
  - **Zero or more modifiers, such as public or private**
  - **The field's type**
  - **The field's name**
  - **Example:- Look at Bicycle class**
- The public keyword identifies these fields as public members, accessible by any object that can access the class.

# Anatomy of a method

Class methods are very much like functions. Methods include a visibility modifier, return type, method name, and optional parameters



# Defining Methods

```
public float calculateSimpleInterest(
        float pri, float rate, float time){

        return((pri*rate*time)/100));
}
```

Method declarations have six components:

- **Modifiers** — such as public, private
- **Return type** — the data type of the value returned by the method, or void if the method does not return a value
- **Method name**
- **Parameter list in parenthesis** — a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses
- **An exception list** — will be discussed later
- **The method body, enclosed between braces** — the method's code, including the declaration of local variables, goes here

# Access Modifiers

- Access modifiers lets you control what other classes have access to the fields of your class
  - **public modifier** – the field is accessible from all the classes
  - **private modifier** – the field is accessible only with in its own class

In the spirit of encapsulation, it is common to make fields private. We still need access to the fields, however. This can be done *indirectly* by adding public methods that obtain the field values for us

# Example

```
public class Bicycle {

   public int cadence = 0;
   public int speed = 0;
   public int gear = 1;

   public void changeCadence(int
   newValue){
        cadence = newValue;
   }
   public void changeGear(int newValue)
   {
        gear = newValue;
   }
   public void speedUp(int increment) {
        speed = speed + increment;
   }
   public void applyBrakes(int
   decrement) {
        speed = speed - decrement;
   }
} // End of Bicycle
```

# Example

```
public class Bicycle {
   private int cadence = 0;
   private int speed = 0;
   private int gear = 1;

   public void changeCadence(int newValue) {
        cadence = newValue;
   }
   public void changeGear(int newValue) {
        gear = newValue;
   }
   public void speedUp(int increment) {
        speed = speed + increment;
   }
   public void applyBrakes(int decrement) {
        speed = speed - decrement;
   }
} // End of Bicycle
```

# Naming a method

- **By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized**
  - **run**
  - **runFast**
  - **getBackground**
  - **getFinalData**
  - **compareTo**
  - **setX**
  - **isEmpty**

A method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

# Method Overloading

- The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists
- Overloaded methods are differentiated by the number and the type of the arguments passed into the method
- You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.
- The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

# Method Overloading

Different Method Signature → Overloading

Case-1 – No. of Parameters are different

Example–

```
Public int add (int a, int b)
{
  - - -
}

Public int add (int a, int b, int c)
{
  - - -
}
```

# Method Overloading

Case 2 - Different Datatype of Parameters

```
Public int add( int a, int b, int c)
  {
    ----
  }

Public double add (double a, double b, double c)
  {
    ----
  }
```

# Method Overloading

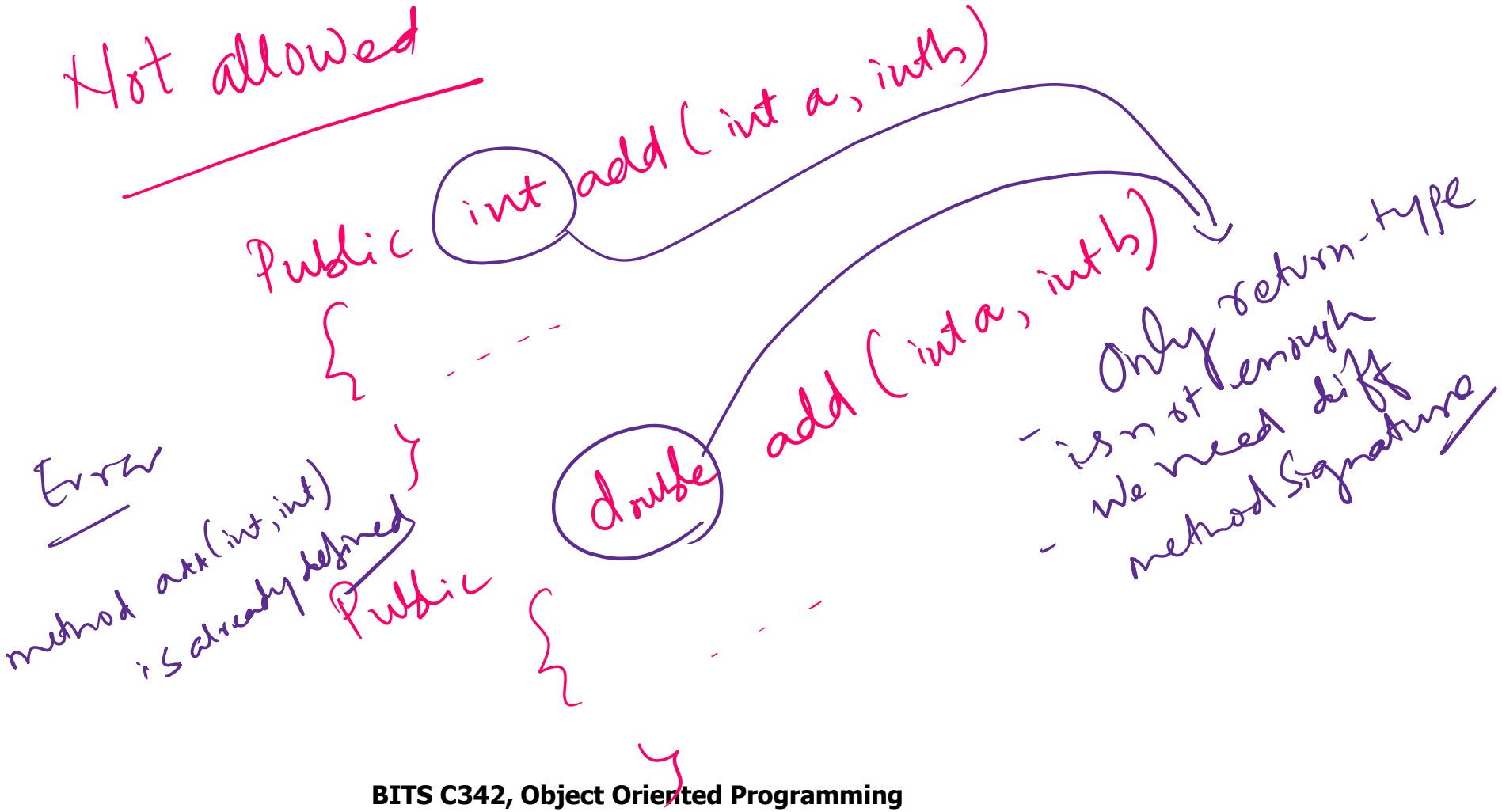
Case 3 – Order of Parameters is different

Example

```
Public Void Compute ( String name, int id)
{ - - -
}

Public Void Compute (int id, String name)
{ - - -
}
```

# Method Overloading



# Method Overloading – Example

```
public class Adder {

   public int addInteger(int x, int y){
        return(x+y);
   }

   public float addFloat(float x, float y){
        return(x+y);
   }

   public String addString(String x, String y){
        return(x+y);
   }

   public double addDouble(double x, double y){
        return(x+y);
   }
}
```

# Method Overloading – Example

```
public class AdderDemo {

        public static void main(String args[]){
                int sum = 0, x = 5, y = 10;
                Adder adder = new Adder();
                sum = adder.addInteger(x,y);
        }

}
```

# Adder class – overloaded methods

```
public class Adder {

   public int add(int x, int y){
        return(x+y);
   }

   public float add(float x, float y){
        return(x+y);
   }

   public String add(String x, String y){
        return(x+y);
   }

   public double add(double x, double y){
        return(x+y);
   }
}
```

# Method Overloading – Example

```
public class AdderDemo {

        public static void main(String args[]){
                int sum = 0, x = 5, y = 10;
                Adder adder = new Adder();
                sum = adder.add(x,y);
        }
}
```

# Adder class - Revisited

**Adder is a stateless class. There is no point in creating a instance of such a class. The methods can be marked *public static*.**

```
public class Adder {

   public static int addInteger(int x, int y){
        return(x+y);
   }

   public static float addFloat(float x, float y){
        return(x+y);
   }

   public static String addString(String x, String y){
        return(x+y);
   }

   public static double addDouble(double x, double y){
        return(x+y);
   }
}
```

# Adder class – overloaded methods



It can be observed that even static methods can be overloaded

```
public class GenericAdder {

    public static int add(int x, int y){
        return(x+y);
    }

    public static float add(float x, float y){
        return(x+y);
    }

    public static String add(String x, String y){
        return(x+y);
    }

    public static double add(double x, double y){
        return(x+y);
    }
}
```

# Method Overloading – Example

```
public class AdderDemo {
   public static void main(String args[]){
        int sum = 0, x = 5, y = 10;

        //Not the right way of making a call to static method
        //GenericAdder genAdder = new GenericAdder();
        //sum = genAdder.add(x,y);

        //a call to the static method
        //should be made in static way
        sum = GenericAdder.add(x,y);
   }
}
```

# Providing constructors

- A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

```
public Bicycle(int startCadence, int startSpeed,
                                 int startGear){
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

# Providing constructors

- Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
  gear = 1;
  cadence = 10;
  speed = 0;
}

Bicycle yourBike = new Bicycle();
```

Invokes the no-argument constructor to create a new Bicycle object called yourBike.

# Providing constructors

- Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

- You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors.

- You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

# Passing Information to a Method or a Constructor

```
public double computePayment(double loanAmt, double rate,
                double futureValue, int numPeriods){

        //your code goes here
}
```

***Parameters*** refers to the list of variables in a method declaration. ***Arguments*** are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

# Parameter Types

- You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the computePayment method, and reference data types, such as objects and arrays.

- This method that accepts three Point objects as argument. The method creates a new Triangle object and initializes it from three Point objects (assume that Point is a class that represents an x, y coordinate)

```
public Triangle triangleFrom(Point p1, Point p1,
                                        Point p1) {
    // method body goes here
}
```

**The Java programming language doesn't let you pass methods into methods. But you can pass an object into a method and then invoke the object's methods.**

# Parameter Names

- When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

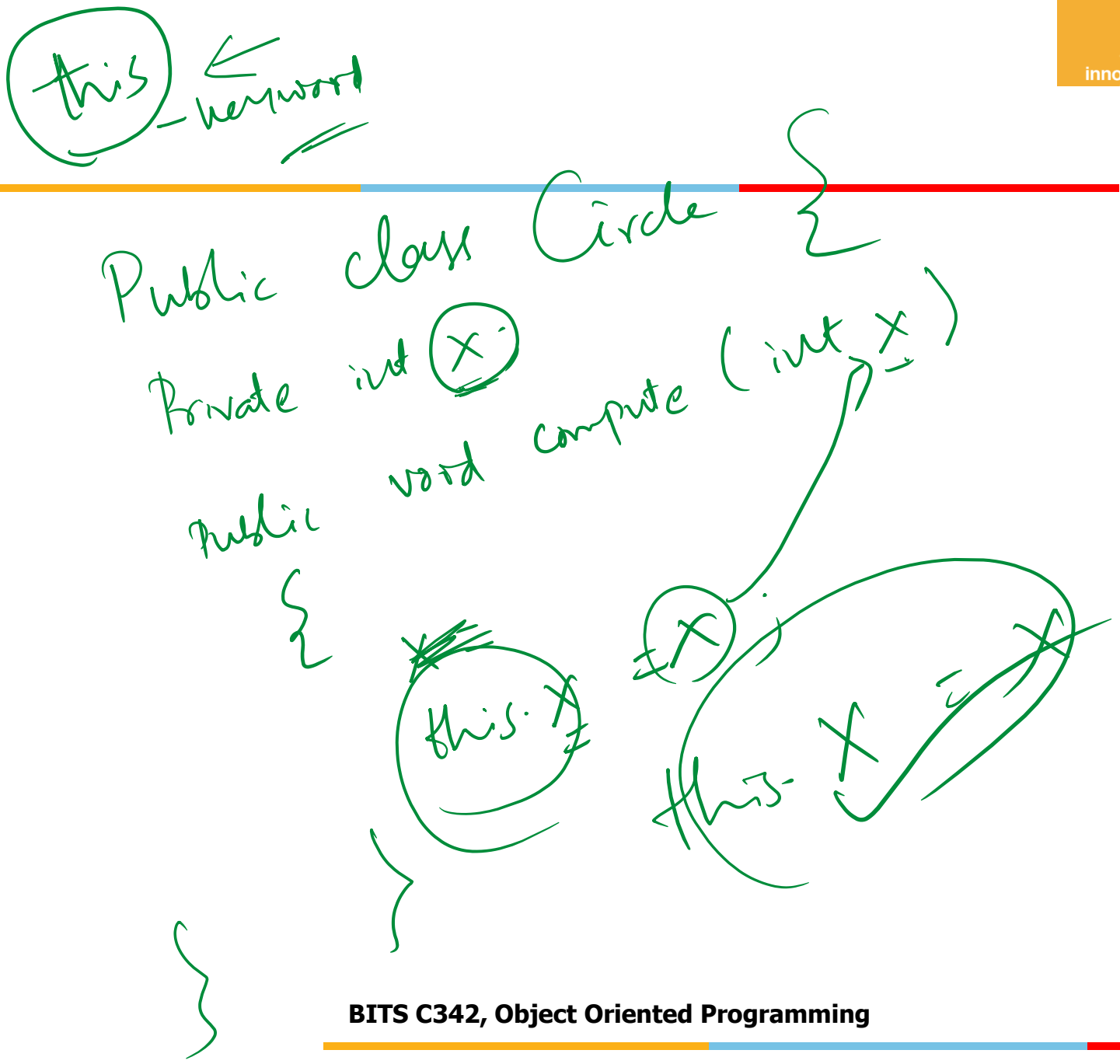
- The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

- A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field.

# Parameter Names

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

- The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, not to the field. To access the field, you must use a qualified name.


this
keyword
Public class Circle
Private int x;
public void compute (int x)
this.x
x
this.x

# Passing Primitive Data Type Arguments

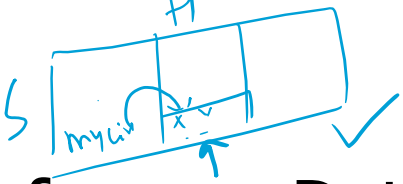
```
public class PassPrimitiveByValue{

    public static void main(String[] args) {

        int x = 3;
        passMethod(x);
        S.o.p("After invoking passMethod, x = " + x);
    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

**Primitive arguments, such as an int or a double, are passed into methods by value. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.**





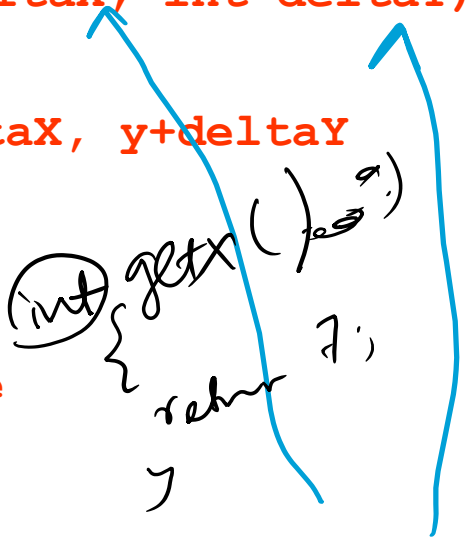
# Passing Reference Data Type Arguments

- Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {

   // code to move origin of circle to x+deltaX, y+deltaY
   circle.setX(circle.getX() + deltaX);
   circle.setY(circle.getY() + deltaY);

   //code to assign a new reference to circle
   circle = new Circle(0, 0);
}
```



**INVOKE: moveCircle(myCircle, 23, 56) ;**

# Passing Reference Data Type Arguments

- Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively. These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.

[Lecture Slides on Reference Types & Java Memory Management have detailed discussion on this topic]

# THANK YOU