Meeting 1: Objects and Classes

Edited by: Dr. Nadine Zbib

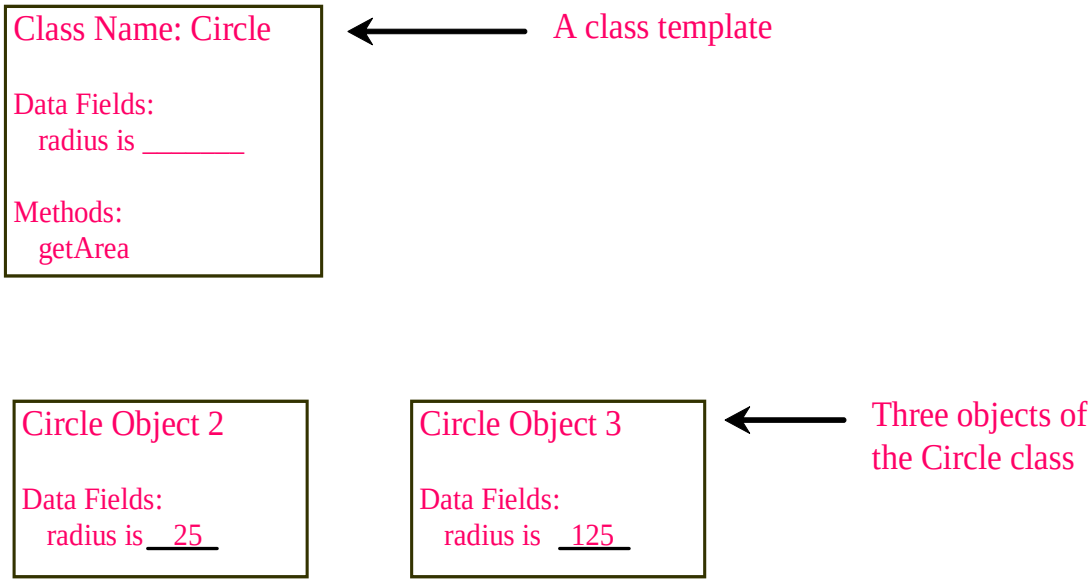
Objectives

- To describe, declare, create objects and classes.
- To create objects using constructors.
- To access objects via reference variables.
- To distinguish between instance and static variables and methods.
- To define private data fields with appropriate **get** and **set** methods.

O.O. Programming Concepts

- Object-oriented programming (OOP) involves programming using objects.
- **An *object*** represents an entity in the real world that can be distinctly identified.
- For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors.
- **The *state* of an object** consists of a set of *data fields* (also known as *properties*, *attributes*) with their current values.
- **The *behavior* of an object** is defined by a set of methods.

Objects



An object has both a state and behavior. The state defines the object (current values of its attributes), and the behavior defines what the object does.

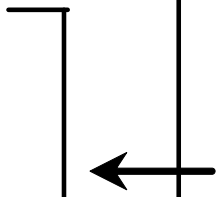
Classes

- ***Classes*** are constructs that define objects of the same type. A class is used to group related objects together. A class is considered as a factory to create objects.
- A Java class uses variables (attributes) to define data fields and methods to define behaviors.
- Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

An Example of Classes

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Zero-argument-Construct */
  Circle() {
  }

  /** One-argument Construct  */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}//end of a class
```

Data field

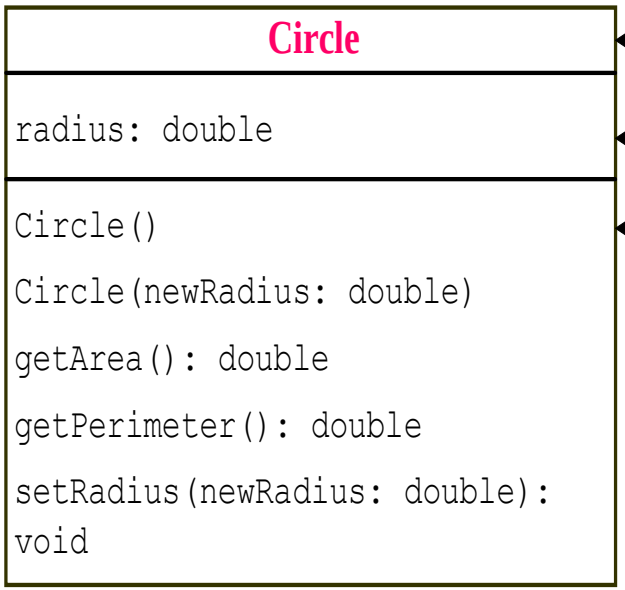


Constructors

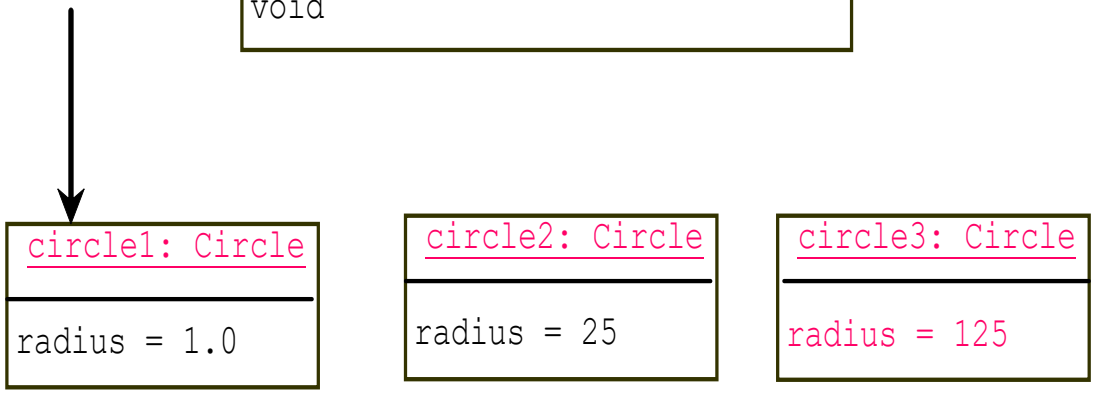
Method

UML Class Diagram

UML Class Diagram



UML notation for objects



Unified Modeling Language (UML): is a standard modeling language, which is based on diagrams and their construction, meaning and use.

Example: Defining Classes and Creating Objects

Objective: Demonstrate creating objects, accessing data, and using methods.

```
public class TestCircle {
    public static void main(String[] args) {
    // Create a circle with radius 1
    Circle circle1 = new Circle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    Circle circle2 = new Circle(25);
    System.out.println("The area of the circle of radius "+ circle2.radius +" is"
    + circle2.getArea());
    // Create a circle with radius 125
    Circle circle3 = new Circle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }
}//end of class
```

TestSimpleCircle

Run

Constructors

- Constructors are a special kind of methods that are invoked to initialise objects.

```
//Zero-argument constructor
Circle() {
}

//One-argument constructor
Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors, cont.

- A constructor with no parameters is referred to as a *no-arg or zero-arg constructor*.
- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Default Constructor

- A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class.

- This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

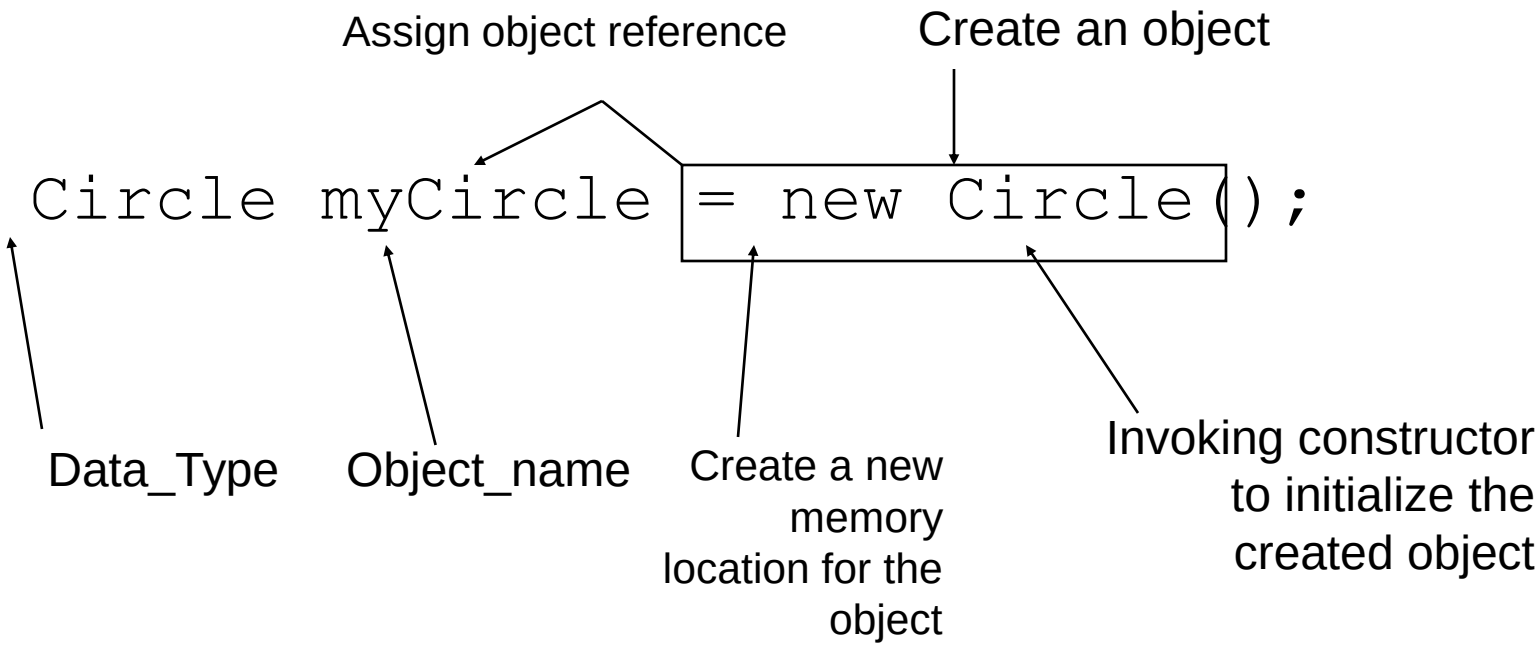
Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

```
ClassName objectRefVar = new ClassName();
```

- Example:



Accessing Class Members

There are two ways to access the class members:

(1) **Using a qualified name with an object reference and a dot notation,**

for example: `objectRefVar.methodName(arguments)`

e.g., `myCircle.getArea();`

(2) **Using a simple name a name without a reference and dot notation,**

e.g., `getArea();` or `radius = 1;`

When to use them?

Simple name:

- Class members could be accessed using the simple name within their class.
- If we say a member is inherited we mean that the subclass has the same access to it as if it were defined in the subclass.

Qualified name:

Other classes would have to have to use **a qualified name**.

animation

Trace Code

Declare myCircle

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

myCircle

no value

animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

myCircle | no value

: Circle

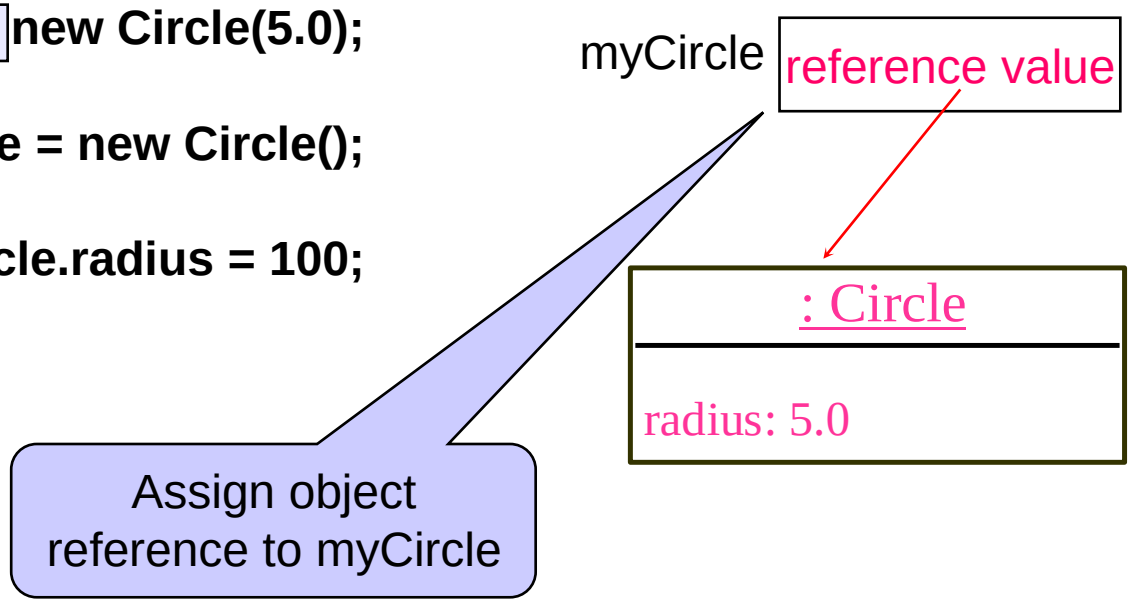
radius: 5.0

Create a circle

animation

Trace Code, cont.

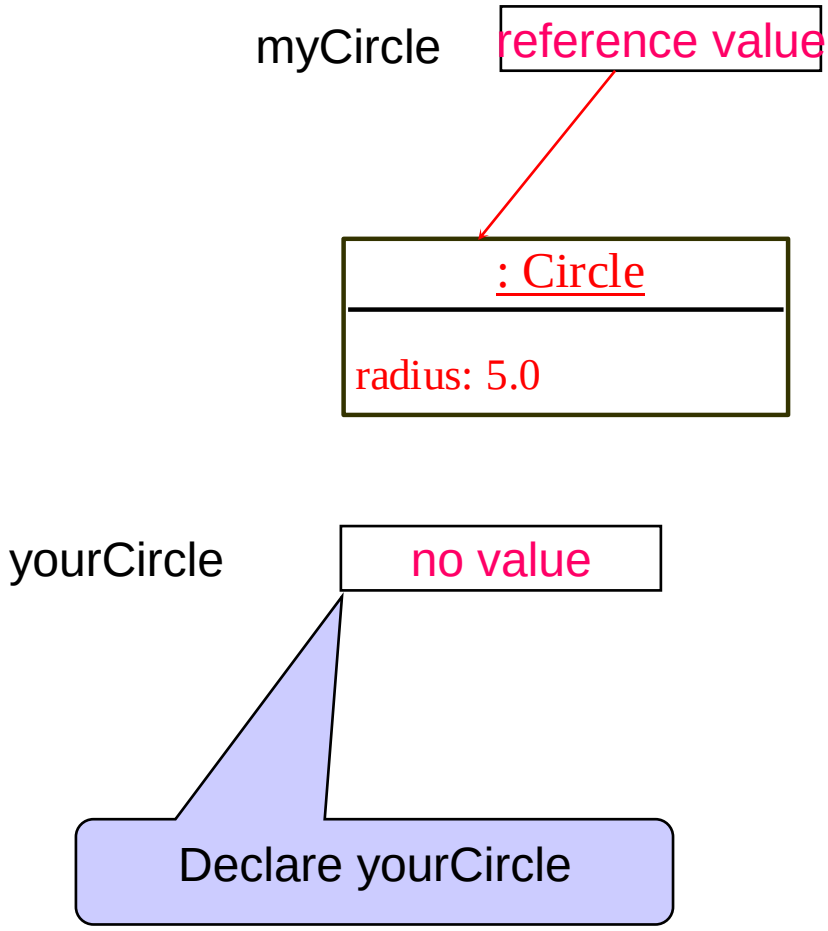
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



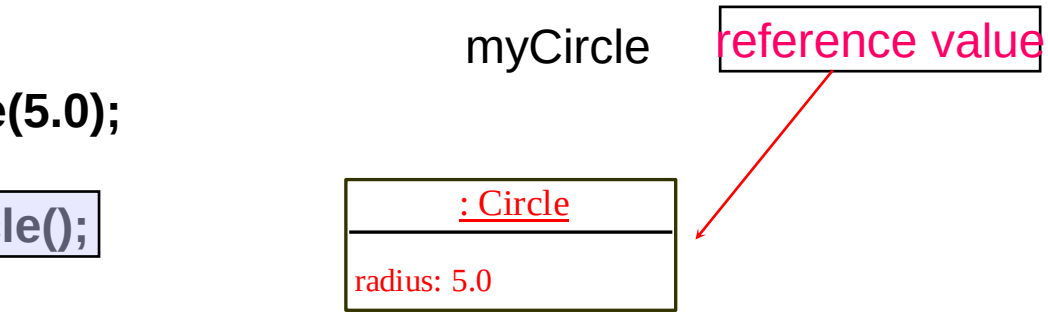
animation

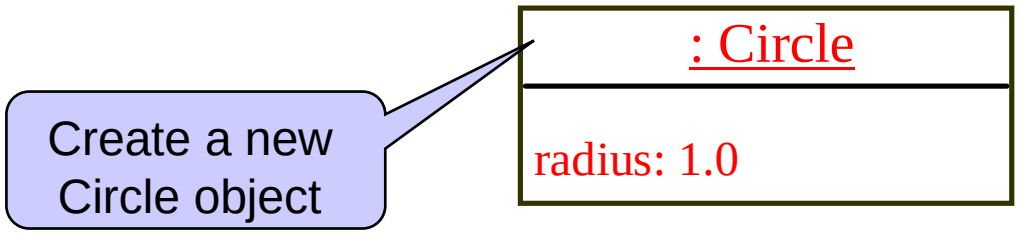
Trace Code, cont.

Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

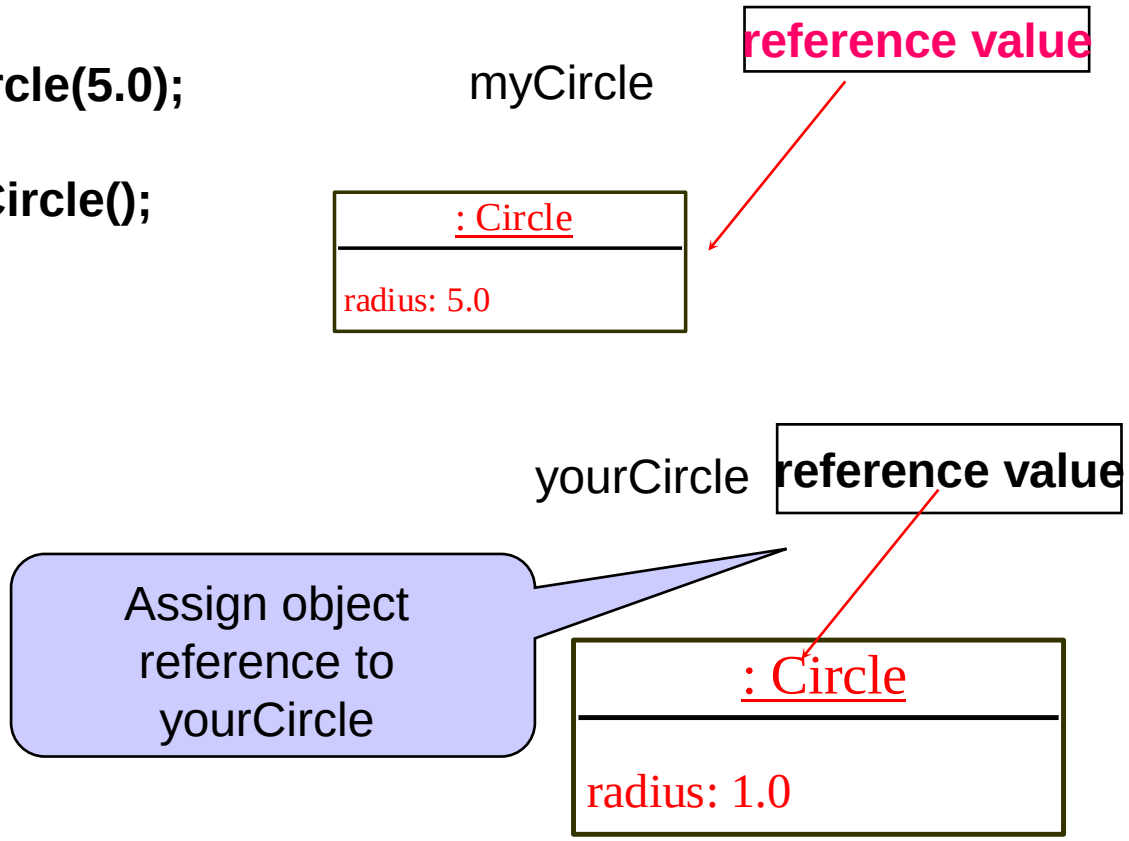
yourCircle.radius = 100;





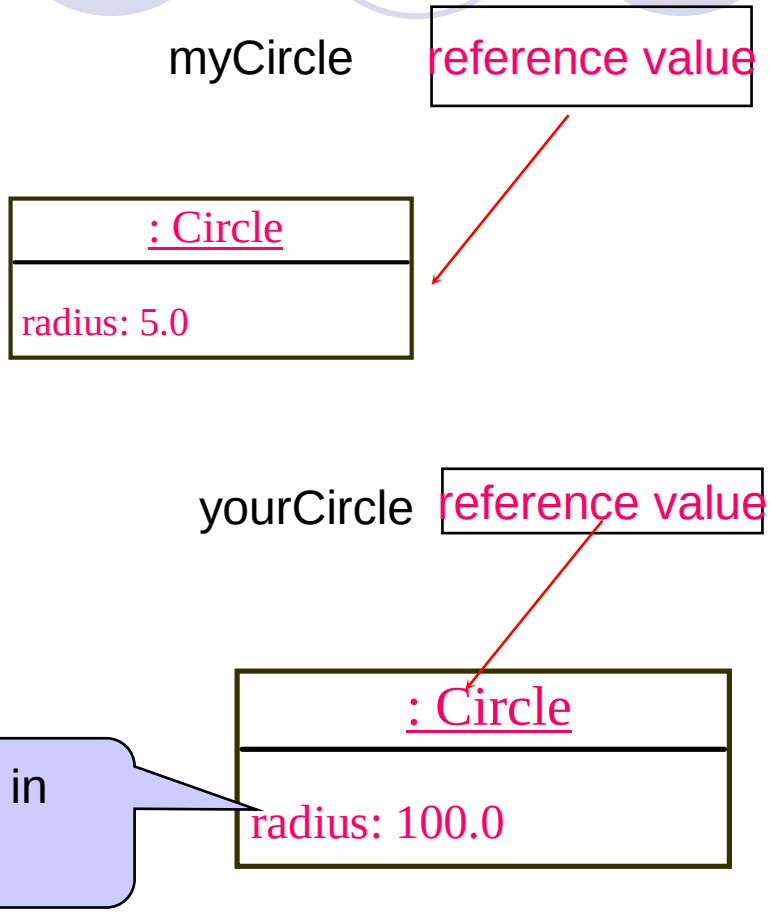
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Caution

Recall that you use

Math.methodName(arguments) (e.g., Math.pow(3, 2.5))

to invoke a method in the Math class. Can you invoke getArea() using SimpleCircle.getArea()? The answer is no.

All the methods used before in M105 module were static methods, which are defined using the static keyword.

However, getArea() is non-static. It must be invoked from an object using

```
objectRefVar.methodName(arguments) (e.g.,
  myCircle.getArea()).
```

Data fields (Attributes)

The data fields can be of primitive types as any other variables or reference types.

For example, the following Student class contains a data field name of the String type.

```
public class Student {
  String name;    // name has default value null
  int age;        // age has default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
 char gender;    // c has default value '\u0000'
}
```

The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

Default Value for a Data Field

The default value of a data field is:

- null for a reference type,
- 0 for a numeric type,
- false for a boolean type,
- and '\u0000' for a char type.

However, Java assigns no default value to a local variable inside a method.

Creating Objects from student Class

```
public class Test {

  public static void main(String[] args) {

   Student student = new Student();

    System.out.println("name? " + student.name);

    System.out.println("age? " + student.age);

    System.out.println("isScienceMajor? " + student.isScienceMajor);

    System.out.println("gender? " + student.gender);

  }

}// End of class
```

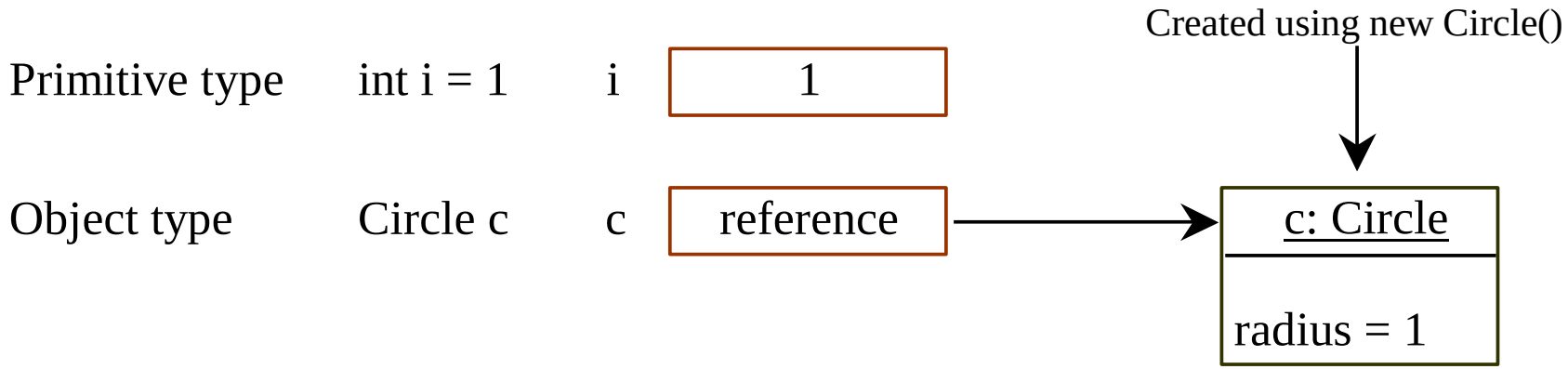
Example

Java assigns default values for attributes, but NO default valuee to local variables inside a method.

```
public class Test {

  public static void main(String[] args) {

    int x; // x has no default value

    String y; // y has no default value

    System.out.println("x is " + x);

    System.out.println("y is " + y);

  }

}
```

Compile error: variable not initialized

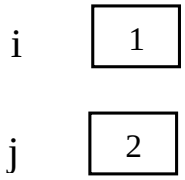
Differences between Variables of Primitive Data Types and Object (Reference) Types



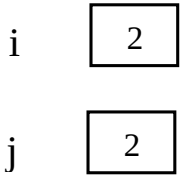
Copying Variables of Primitive Data Types and Object (reference) Types

Primitive type assignment: i = j

Before:

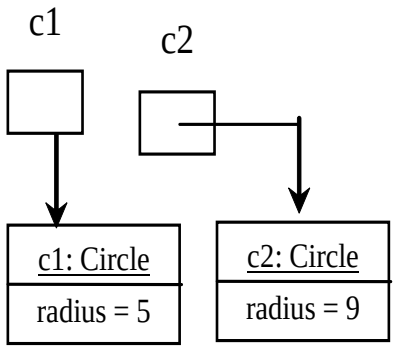


After:

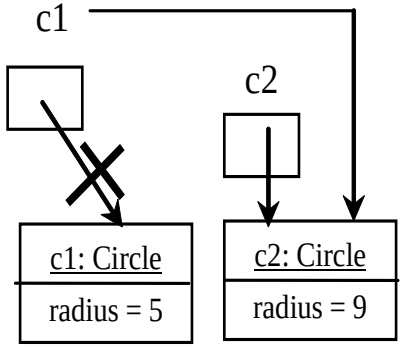


Object type assignment: c1 = c2

Before:



After:



Garbage Collection

As shown in the previous figure, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. Garbage is automatically collected by (java Virtual Machine (JVM).

Garbage Collection, cont

TIP: If you know that an object is no longer needed, you can explicitly assign null to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any variable.

Data fields and Methods

Data fields can be classified into:

- Instance variables and static variables

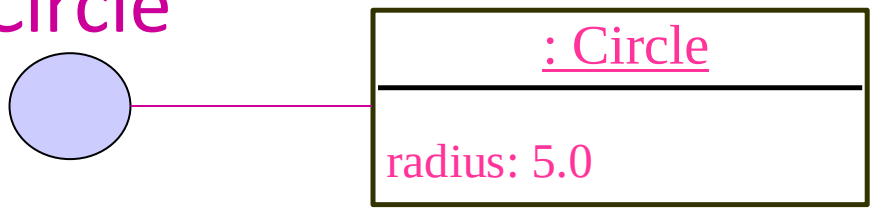
Methods can be classified into:

- Instance methods and static methods

Instance Variables and Instance Methods

- Instance variables belong to a specific instance. For example:

```
Circle myCircle = new Circle(5.0);
```



- Instance methods are invoked by an instance of the class. For example:

```
double x = myCircle.getArea();
```

Static Variables, Constants, and Methods

- Static variables (class variables) are shared by all the instances of the class.
- Class variables are used to keep count of the number of objects of a class type that have been created. As each object would have to update the count every time a new object came into existence – (in the constructors).

- Static methods (class methods) are not tied to a specific object.

- Static constants are final variables shared by all the instances of the class.

- You can access class variables and methods using class name.

Static Variable Example

```
Public class VariableDemo {
   static int count=0;

   public void increment() { count++; }

   public static void main(String args[]) {
       VariableDemo obj1=new VariableDemo();
       VariableDemo obj2=new VariableDemo();
       obj1.increment();
       obj2.increment();
       System.out.println("Obj1: count is="+obj1.count);
       System.out.println("Obj2: count is="+obj2.count);
   }
}
```

Output:

```
   Obj1: count is=2 Obj2: count is=2
```

Static Constants

To declare static variables, constants, and methods, use the static modifier.

For example: PI is declared inside Math class as constant:

```
public static final double PI = 3.14159265358979323846;
```

So you can access it using: `Math.PI`

- It is public because we wish it to be available to other classes,
- It is static because it has one value for all objects of the class
- It is final because pi is a constant.
- It is double because that is the most accurate decimal representation we can use.

Static methods (also known as a class methods) and their uses

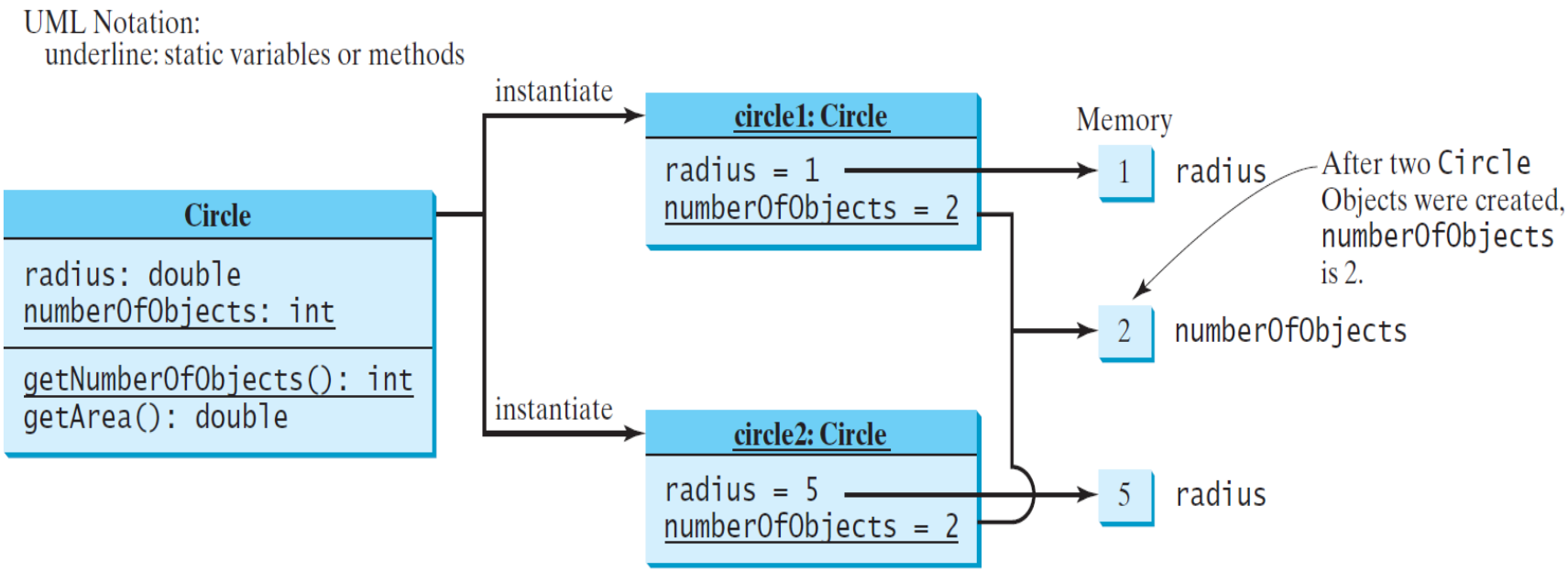
- Static methods carry out general functions not associated with objects.
- **Static Variable can be accessed directly in a static method.**

– For example, the static method **max within the class Math** returns the greater of its two arguments. The method header for one version of max looks like this:

```
public static double max (double a, double b)
{
   if (a>b)
      return a;
   else
      return b;
}
```

- To access it: **double val = Math.max(Math.PI, 4.0);**
- The **main method is** a static method.

Static Variables, Constants, and Methods, cont.



Packages

- A package is used to associate a number of classes that are closely related to one another.
  - if classes form a cohesive group, they should be marked as belonging to the same package.
  - For example, Java has a collection of classes named java.math.
  - If we wanted to make use of this package, we would add a statement to advise the Java system of this, by adding an **import statement at the** beginning of our program. In this case we would say:

```
import java.math.*;
public class User
{
    // etc.
}
```

The .* notation means 'all the classes in the package.

Access modifiers

There are three keywords associated with controlling levels of access to class members in Java: **public, protected & private. These are known as access modifiers.**

There is actually a fourth level of access, 'default'. There is no 'default' keyword; default access arises when none of the access modifiers is specified.

- **`public`**

  The class, data, or method is visible to any class in any package using qualified name (a reference and a dot notation).

- **`private`**

  The data or methods can be accessed only by the declaring class. The get and set methods are used to read and modify private properties.

- **`protected and default access will be discussed later`**

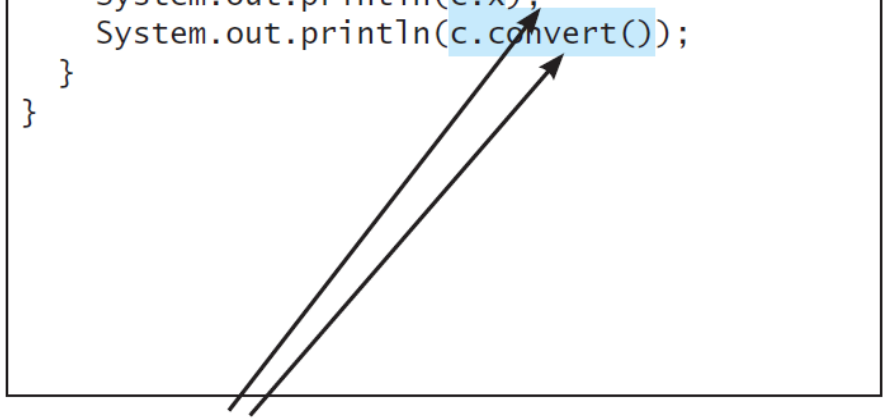
NOTE

An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object `c` is used inside the class `C`.

```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```



(b) This is wrong because `x` and `convert` are private in class `C`.

Why Data Fields Should Be private?

- To protect data.
- To make code easy to maintain.
- To achieve encapsulation principle (declare attributes as private and methods as public) and information hiding which are of Object Oriented programming features.

Getter and setter methods

Usually you need to declare a setter and a getter methods for each instance variable.

A getter method (also called **an 'accessor' method**):

– It is used to access private attributes from outside the class where they were declared.

– A getter method should not change the state of the object.

– All private attributes should have getter methods .

For example, inside the class Circle, we need to add another method:

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

In the main method: we invoke `getRadius()` using object name.

Setter Method

A setter method (also called a 'mutator' method)

- It is one that changes the state of an object by setting the value of an instance variable.
- Setter methods do not normally return a value.

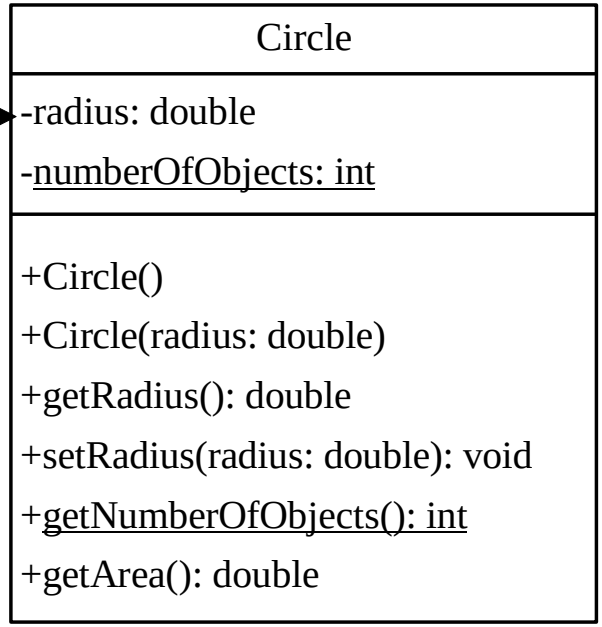
For Example: Inside Circle class we need to add:

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

In the main method: we invoke `setRadius()` using object name.

Example of Data Field Encapsulation

The - sign indicates private modifier



The radius of this circle (default: 1.0).

The number of circle objects created.

Constructs a default circle object.

Constructs a circle object with the specified radius.

Returns the radius of this circle.

Sets a new radius for this circle.

Returns the number of circle objects created.

Returns the area of this circle.

CircleWithPrivateDataFields

TestCircleWithPrivateDataFields

Run

Passing Objects to Methods

- Passing by value for primitive type value
  (the value is passed to the parameter)
- Passing by value for reference type value
  (the value is the reference to the object)

TestPassObject

Run

Passing Objects to methods example

```
public class TestPassObject {
  public static void main() {
    Circle c1 = new Circle(1);
   // Print areas for radius 1, 2, 3, 4, and 5.
    int n = 5;
    printAreas(c1, n);
    System.out.println("\n" + "Radius is " +
    c1.getRadius());
    System.out.println("n is " + n);
  }
   public static void printAreas( Circle a, int times) {
     System.out.println("Radius \t\tArea");
     while (times >= 1) {
        System.out.println(a.getRadius() + "\t\t" +
        a.getArea());
        a.setRadius(a.getRadius() + 1);
        times--; }
   }
}// End Class
```

The keyword this

The this keyword is the name of a reference that refers to an object itself.

One common use of the this keyword is reference a class's *hidden data fields*.

Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

Using this to refer to the Hidden Data Fields

```
public class F {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    F.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of F.
F f1 = new F(); F f2 = new F();

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

```
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
      this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}
```

Instance variables-Attributes-

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor in the same class

Every instance variable belongs to an instance represented by this, which is normally omitted

The keyword `this`

- The word **`this`** can be used to **invoke a constructor**.
- You can think of **`this`** as a reference to the class in which it appears.
- Consider the example on the right:

```
public class ThreeInt{
    private int p, q, r;
    public ThreeInt (){
        this(0, 0, 0);
    }
    public ThreeInt (int a){
        this(a, 0, 0);
    }
    public ThreeInt (int a, int b){
        this(a, b, 0);
    }
    public ThreeInt (int a, int b, int c){
        p = a; q = b; r = c;
    }
    // methods associated with ThreeInt
}
```

- The first three constructors **make use of the three-argument constructor**. This is a way of reusing constructors you have written and can **assist with readability** when you have several constructors.
- Do not think of this(a, b, 0) as a call to a method. It is only possible to invoke constructors in this way.
- a constructor cannot invoke itself (whereas a method can).

Array of Objects

```
Circle[] circleArray = new Circle[10];
```

The above statement creates the array which can hold references to ten Circle objects.

It doesn't create the circle objects themselves. They have to be created separately using the constructor of the Student class.

For example, if you try to write:

```
circleArray[0].setRadius(10);
```

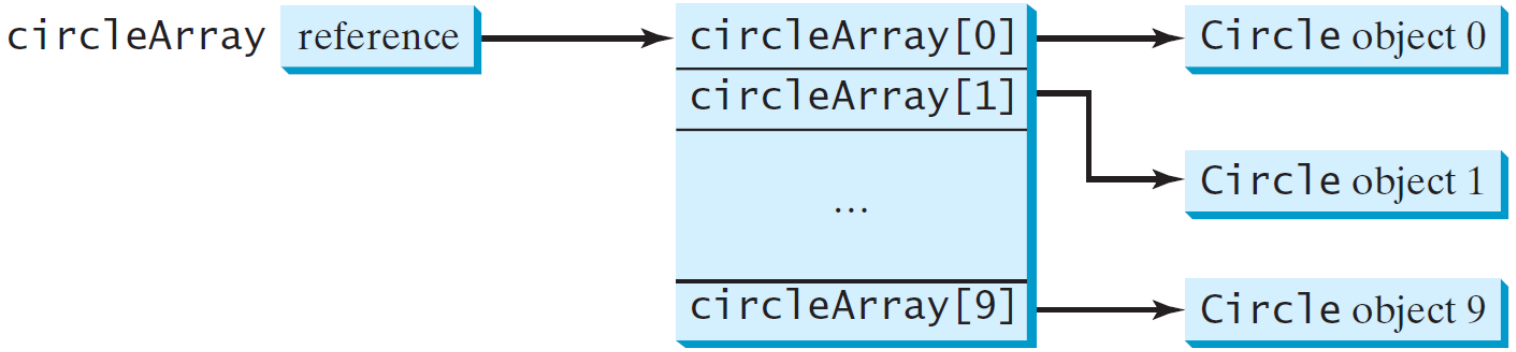
This will produce a run time error since `circleArray[0]` points to null.

Array of Objects, cont.

The Circle objects have to be instantiated using the constructor of the Circle class and their references should be assigned to the array elements in the following way.

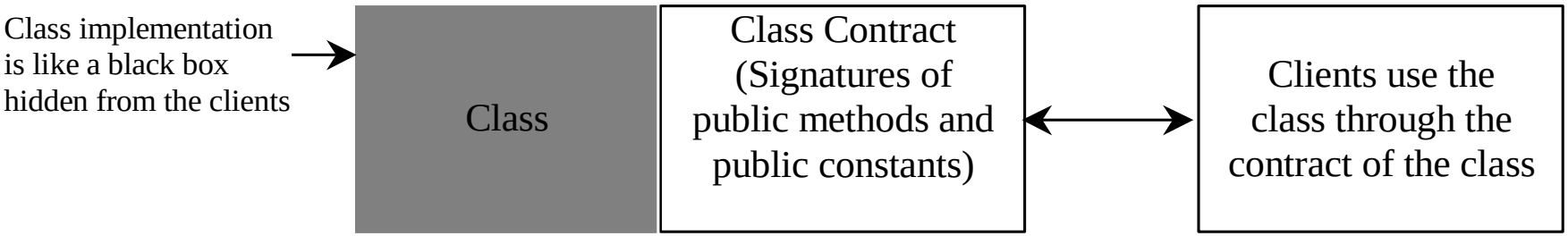
```
for(int i=0; i<circleArray.length; i++)
        circleArray[i] = new circle();
```

```
//Invoking methods
    double x = circleArray[0].getRadius();
```



Class Abstraction and Encapsulation

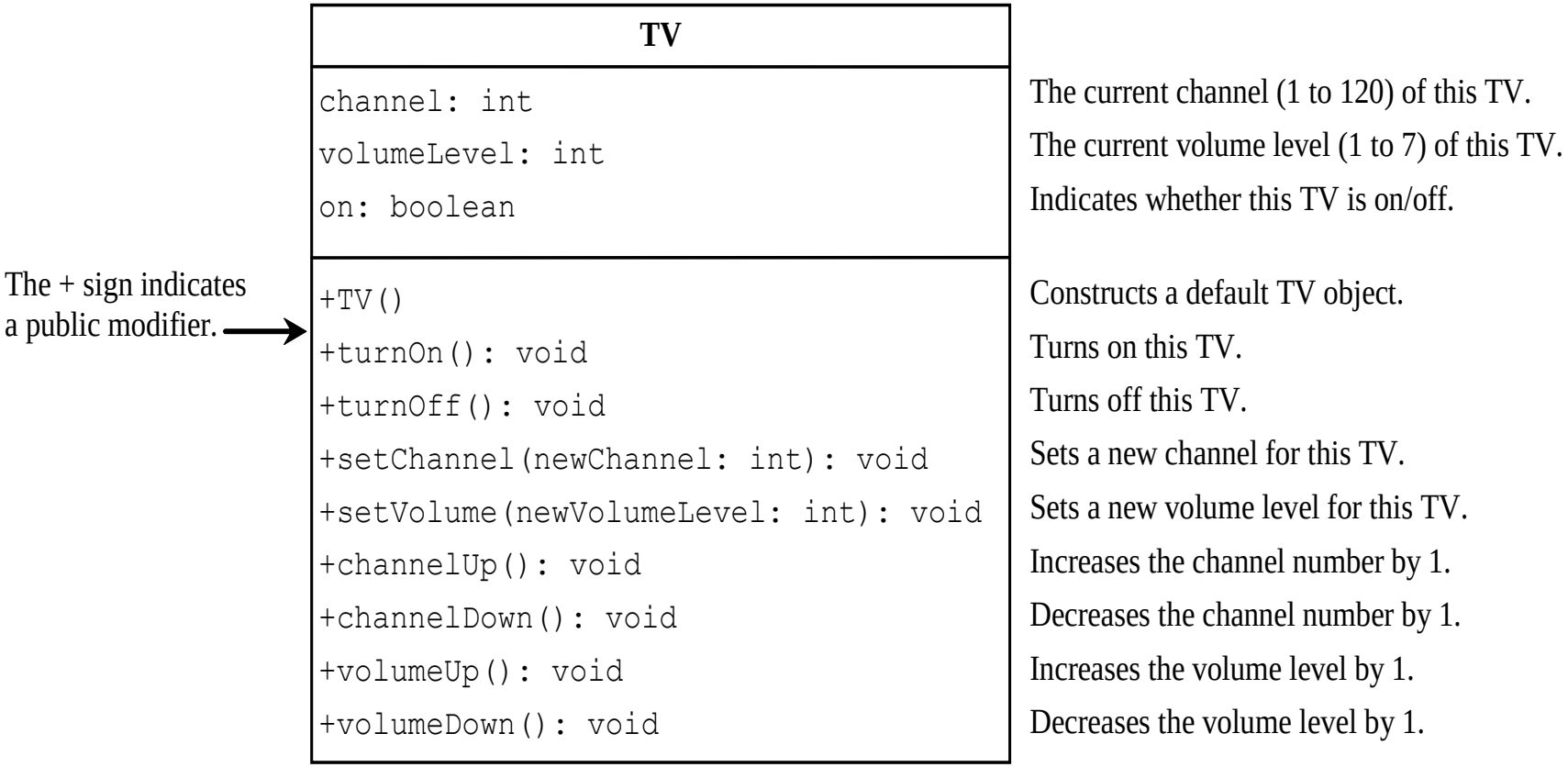
- Class abstraction means to separate class implementation from the use of the class.
- The creator of the class provides a description of the class and let the user know how the class can be used.
- The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Object-Oriented Thinking

- You studied before the fundamental programming techniques for problem solving using loops, methods, and arrays.
- The studies of these techniques lay a solid foundation for object-oriented programming. All the previous code was written in the main method so you could not re-use it in other projects.
- Classes provide more flexibility and modularity for building reusable software.

Example: Defining Classes and Creating Objects



Class TV

```
public class TV {
  //Attributes
  int channel = 1; // Default channel is 1
  int volumeLevel = 1; // Default volume level is 1
  boolean on = false; // By default TV is off

 // constructors
  public TV() {}

  //Methods
  public void turnOn() {
    on = true;
  }
  public void turnOff() {
    on = false;
  }
  public void setChannel(int newChannel) {
    if (on && newChannel >= 1 && newChannel <= 120)
      channel = newChannel;
  }
  public void channelUp() {
    if (on && channel < 120)
      channel++;
  }
```

```
public void setVolume(int
newVolumeLevel) {
    if (on && newVolumeLevel >= 1 &&
          newVolumeLevel <= 7)
      volumeLevel = newVolumeLevel;
  }

  public void channelDown() {
    if (on && channel > 1)
      channel--;
  }

  public void volumeUp() {
    if (on && volumeLevel < 7)
      volumeLevel++;
  }

  public void volumeDown() {
    if (on && volumeLevel > 1)
      volumeLevel--;
  }

}// end of class
```

TV

TestTV

Run

Test TV class

```
public class TestTV {
   public static void main() {
      TV tv1 = new TV();
      tv1.turnOn();
      tv1.setChannel(30);
      tv1.setVolume(3);

      TV tv2 = new TV();
      tv2.turnOn();
      tv2.channelUp();
      tv2.channelUp();
      tv2.volumeUp();
      System.out.println("tv1's channel is " +     tv1.channel +
              " and volume level is " + tv1.volumeLevel);
      System.out.println("tv2's channel is " + tv2.channel       +
              " and volume level is " + tv2.volumeLevel);
   }
}
```

ArrayList Class

- You can create an array to store objects. But the array's size is fixed once the array is created.
- Java provides the ArrayList class that can be used to store an unlimited number of objects.

ArrayList Object

- ArrayList is an indexable **ordered collection** of variable size that permits duplicate elements.
- **Ordered collection means** each element in the collection has a well-defined place, as indicated by its index number.

ArrayList VS. Array

Objects of the generic class ArrayList are similar to arrays, but have differences.

Similarities:

- Both can store a number of references (elements).
- The data can be accessed via an index.
- Elements duplication is allowed.

ArrayList Vs. Array

Differences:

- ArrayList has no fixed size. You can extend the ArrayList as needed automatically by adding elements.
- The ArrayList object requests more space from the Java run-time system, if necessary.
- The ArrayList is not ideal in its use of memory.
- As soon as an ArrayList requires more space than its current capacity, the system will allocate more memory space for it. The extension is more than required for the items about to be added. So there will often be empty locations within an ArrayList – that is, the size will often be less than the capacity.

Two important terms in connection with ArrayList objects:

- The capacity of an ArrayList object is the maximum number of items it can currently hold.
- The size of an ArrayList object is the current number of items stored in the ArrayList.

The ArrayList Class

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first match of element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : E	Removes the element at the specified index, and return it.
+set(index: int, o: E) : E	Sets the element at the specified index.

Generic Type

ArrayList is known as a generic class with a generic type E, e.g., **ArrayList<E>.**

You can specify a concrete type to replace E when creating an ArrayList.

For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings. We read it as ArrayList **of** String.

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

TestArrayList Run

ArrayList Declaration and Creation

How to declare an ArrayList?

ArrayList <Object-Type> list1 = new ArrayList <Object-Type> (); .1

ArrayList <Object-Type> list1 = new ArrayList <Object-Type> (Capacity); .2

Note that : you need to use Wrapper Classes instead of primitive Data type inside the angle brackets <…> or any Object-Type.

could be omitted starting Java 7

Example 1:

```
ArrayList <Integer> list2 = new ArrayList <Integer>();
```

- This declaration of an ArrayList specifies that it will hold Integer references
- The type “Integer” can be replaced by any object type.
- This declaration sets up an empty ArrayList .
- The ArrayList capacity is expanded automatically as needed when items are added

ArrayList Declaration and Creation

Example 2:

ArrayList <Double> list1 = new ArrayList <Double> (100);

- This declaration of an ArrayList specifies that it will hold Double references
- The type “Double” can be replaced by any object type.
- This declaration can be used to specify the initial capacity –in this case 100 elements.
- This can be more efficient in avoiding repeated work by the system in extending the list, if you know approximately the required initial capacity.

Exercise(1)

1. Create a list to store cities called cityList.
2. Add the following cities to cityList: London, Paris, Denver, Miami, Tokyo, Seoul.
3. Print out the list contents.
4. Print the size of the cityList.
5. Print out if Miami is in cityList.
6. Print out the location of Denver in the list.
7. Add another city Paris to the list in location 4.
8. Remove the first occurrence of “Paris” from the list.

```
// 1
ArrayList<String> cityList = new ArrayList<>();
// 2
cityList.add("London"); cityList.add("Denver");
cityList.add("Paris"); cityList.add("Miami");
cityList.add("Seoul"); cityList.add("Tokyo");
//3
System.out.println(cityList);

//4-6
System.out.println("List size? " + cityList.size());
System.out.println("Is Miami in the list? "+
cityList.contains("Miami"));
System.out.println("The location of Denver in the
list? "+ cityList.indexOf("Denver"));
//7-8
cityList.add(4, "Paris");
cityList.remove ("Paris");
```

Printing the ArrayList Object

There are 3 ways to print the ArrayList object:

1. System.out.println(cityList);
2. Using for loop as in the array:

```
For(int i=0; i<cityList.size(); i++)
  System.out.prinltn(cityList.get(i));
```

3. Using for each statement

```
for(String x : cityList)
    system.out.println( x);
```

- For each String X belong to cityList, x will hold the value itself not the index.
- The print statement could be replaced by other statement.

Exercise(2)

1. Create a list to store two circles.
2. Add two circles
3. Display the area of the first circle in the list

Solution:

```
ArrayList<Circle> cl = new ArrayList<Circle>();
cl.add(new Circle(5));
cl.add(new Circle(2));
System.out.println("Radius of circle 1= "+ cl.get(0).getRadius());
System.out.println("Area of circle 1 = "+ cl.get(0).getArea());
```

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

DistinctNumbers Run

Array Lists from/to Arrays

It is also possible to create an equivalent list from a array list using the static method Arrays.asList(array).

Example: Creating an ArrayList from an array of objects:

String[ ] array = {**"red"**, **"green", "blue"**};

```
ArrayList<String> list = new ArrayList<String>(Arrays.asList(array));

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

Creating an array of objects from an ArrayList

It is possible to start with a List and create an equivalent array containing the same elements in the same order by using: toArray() method.

```
String[ ] array1 = new String[list.size()];

list.toArray(array1);
```

Collections Utility Class

- Collections class, is a treasure store of class (static) methods for doing things to collections, such as sorting or shuffling them, picking out the largest or smallest item, and so on.
- Collections methods can be used with ArrayList object.
- You need to import java.util.Collections;

Methods of Collections Class

- Reversing:

To reverse a list, pass it as an argument to the reverse() method. The reverse ordered list will be stored in the same argument. For example,

Collections.reverse(cityList);

- Sorting:

To sort a list alphabetically or in ascending order, pass it as an argument to the sort() method. The sorted list will be stored in the same argument For example,

- Collections.sort(cityList);

Methods of Collections Class

Finding the maximum and minimum:

- The maximum or minimum element of a list can be found using the max() and min() methods respectively. These methods return the actual maximum and minimum values, not their positions.
- As with sorting, for this method to succeed, the elements must have a defined ordering, such as alphabetical or numerical.

max and min in an Array List

```
Integer[] array = {8,10,6,3,9};

ArrayList<Integer> l1 = new ArrayList<Integer>(Arrays.asList(array));

System.out.println(Collections.max(l1));

System.out.println(Collections.min(l1));
```

Shuffling an Array List

The elements in a list can be put into random order by passing the list as an argument to the shuffle() method. For example,

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));

Collections.shuffle(list);

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

Exercise(3)

1. Create an array of Integer with values: 90,70,88,66,94,93
2. Create an Integer list name it list1 to have the same values as integer array.
3. Print the list1.
4. Print list1 in reverse order
5. Print out the maximum and minimum number in list1
6. Print out the summation of list1.
7. Rearrange list1 in random order;

Exercise(3)-Solution

```
//1-2
Integer[] a1 = {90,70,88,66,94,93};
ArrayList<Integer> list1 = new ArrayList<Integer>(Arrays.asList(a1));
// 3
        System.out.println(list1);
//4
        Collections.reverse(list1);
        System.out.println(list1);
//5
        System.out.println("Maximum is "  + Collections.max(list1));
        System.out.println("Minimum is  "+ Collections.min(list1));
//6
        double s =0;
        for(Integer y : list1)
                s = s+ y;
        System.out.println("List summation = "+ s);
//7
        Collections.shuffle(list1);
        System.out.println(list1);
```

Please solve the Exercises.