Objects and Classes

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

4. Accessing objects via Reference Variable
   - Reference variables and reference types
   - Accessing an Object's Data and Methods
   - Reference Data Fields and the null value
   - Differences between Variables of Primitive Types and Reference Types
5. Static Variables, Constants and Methods

6. Visibility Modifiers
7. Data Field Encapsulation
8. Passing Objects to Methods

Defining Classes for Objects

What is the name of this class? Any ideas?

Defining Classes for Objects

What is the name of this class? Any ideas?

- Object Oriented Programming;

But what is an object? Any ideas?

Object

An **object** represents an **entity** that can be identified, *e.g.:*

- Student;
- Desk;
- Circle;
- Button,
- Loan

An **object** has a unique:

- State;
- Behaviour.

But what is the **state** of an object? Any ideas?

But what is the **behaviour** of an object? Any ideas?

Lets have a look into these concepts =)

State

State of an object:

- A.k.a. as **attributes**;
- Represented by data fields with their current values;
- Examples:
  - A circle object has an **attribute radius**;
  - A rectangle object has **attributes** width and height;

Behaviour

Behaviour of an object is defined by **methods:**

- To invoke a method on an object is to ask the object to perform an action;
- Example:
  - A circle object may define methods:
    - getArea()
    - getPerimeter()
    - setRadius(radius)

Class

Objects of the same type are defined using a common **class**:

But what is a class? Any ideas?

Class

Objects of the same type are defined using a common **class**:

But what is a class? Any ideas?

Class is a **abstraction** defining (1/3):

- What **attributes** objects should have;
- What **methods** objects should have;

Class

Objects of the same type are defined using a common **class**:

But what is a class? Any ideas?

Class is a **abstraction** defining (2/3):

- An object is an **instance** of a class;
  - An instance is a **concretization** of an **abstraction**;
  - You can create many instances of a class (*i.e.*, **instatiation**);

Class

Objects of the same type are defined using a common **class**:

But what is a class? Any ideas?

Class is a **abstraction** defining (3/3):

- **Constructors**:
  - Invoked to create a new object;
  - Designed to initialize attributes;

Lets try to make these concepts a little bit clearer =)

Example (1/2)

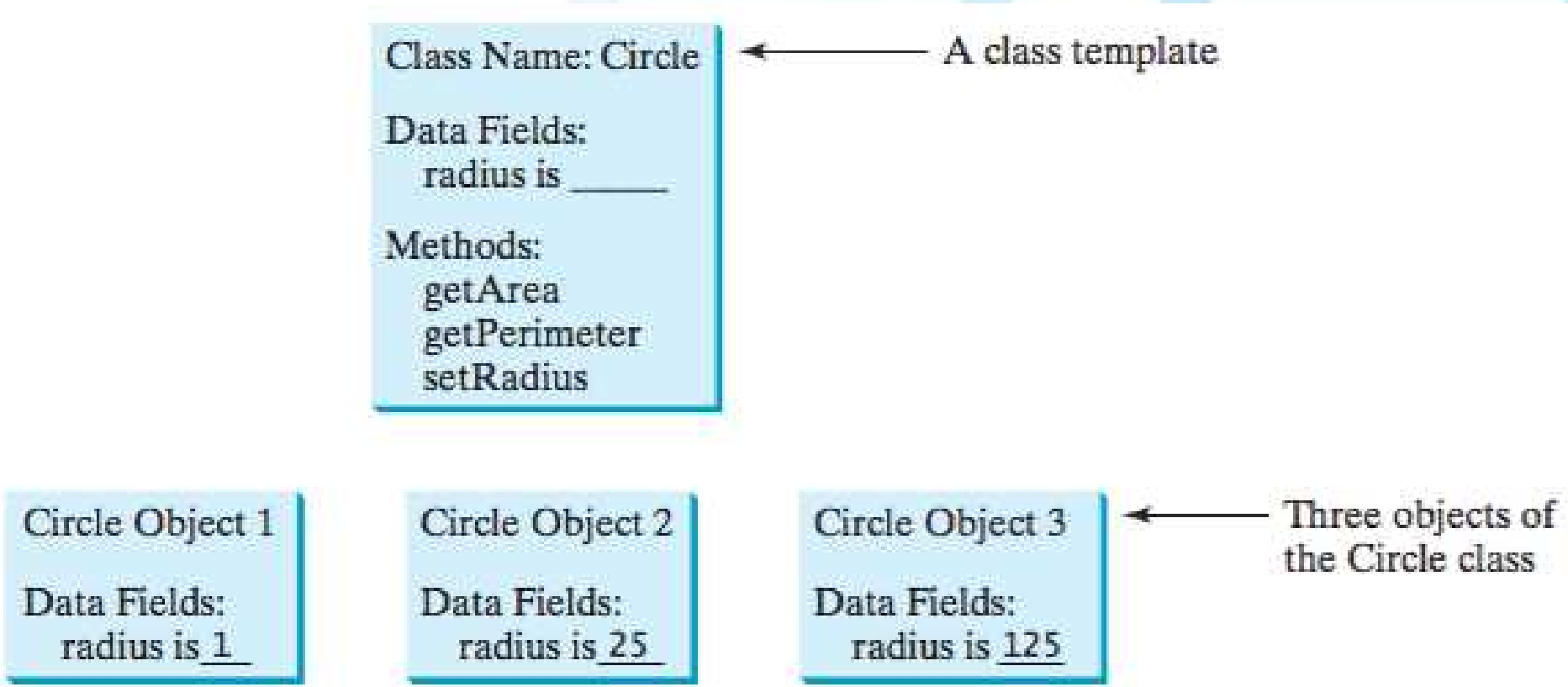


Figure: A class is a template for creating objects.(Source: (Liang, 2014))

Example (2/2)

```
class Circle {
    /** The radius of this circle */
    double radius = 1;

    /** Construct a circle object */
    Circle() { }
      /** Construct a circle object */
    Circle(double newRadius) { radius = newRadius;
    }

     /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }

    /** Return the perimeter of this circle */
    double getPerimeter() {
        return 2 * radius * Math.PI;
    }

      /** Set new radius for this circle */
    double setRadius(double newRadius) {
        radius = newRadius;
    }
}
```

Unified Modeling Language

Class templates and objects can be standardized using Unified Modeling Language (UML) notation.

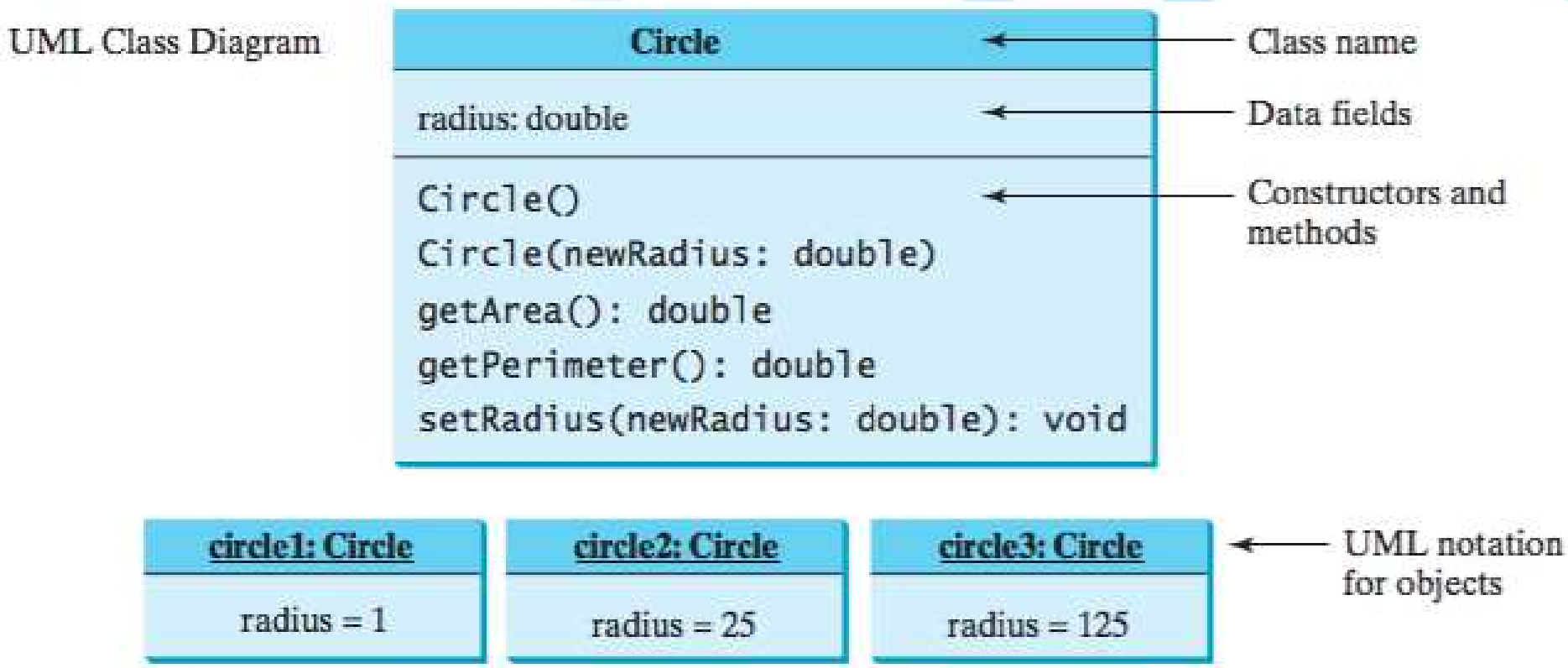


Figure: A class is a template for creating objects.(Source: (Liang, 2014))

Creating Objects

Once the constructor, attributes and methods have been defined:

How can we use it to build a new object? Any ideas?

Creating Objects

Once the constructor, attributes and methods have been defined:

How can we use it to build a **new** object? Any ideas?

- Using the Java keyword: **new** ;)

Example

```
    /** Main method */
    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 v1 — NOT GOOD PRACTICE
    circle2.radius = 100;
     System.out.println("The area of the circle of radius " + circle2.radius + " is " + circle2.getArea());

     // Modify circle radius v2 — GOOD PRACTICE
    circle2.setRadius(100);
    System.out.println("The area of the circle of radius " + circle2.radius + " is " + circle2.getArea());
    }
}
```

TV Exercise (1/4)

Draw the UML diagram for the class television.

Write the Java code for the respective UML diagram.

TV Exercise (2/4)

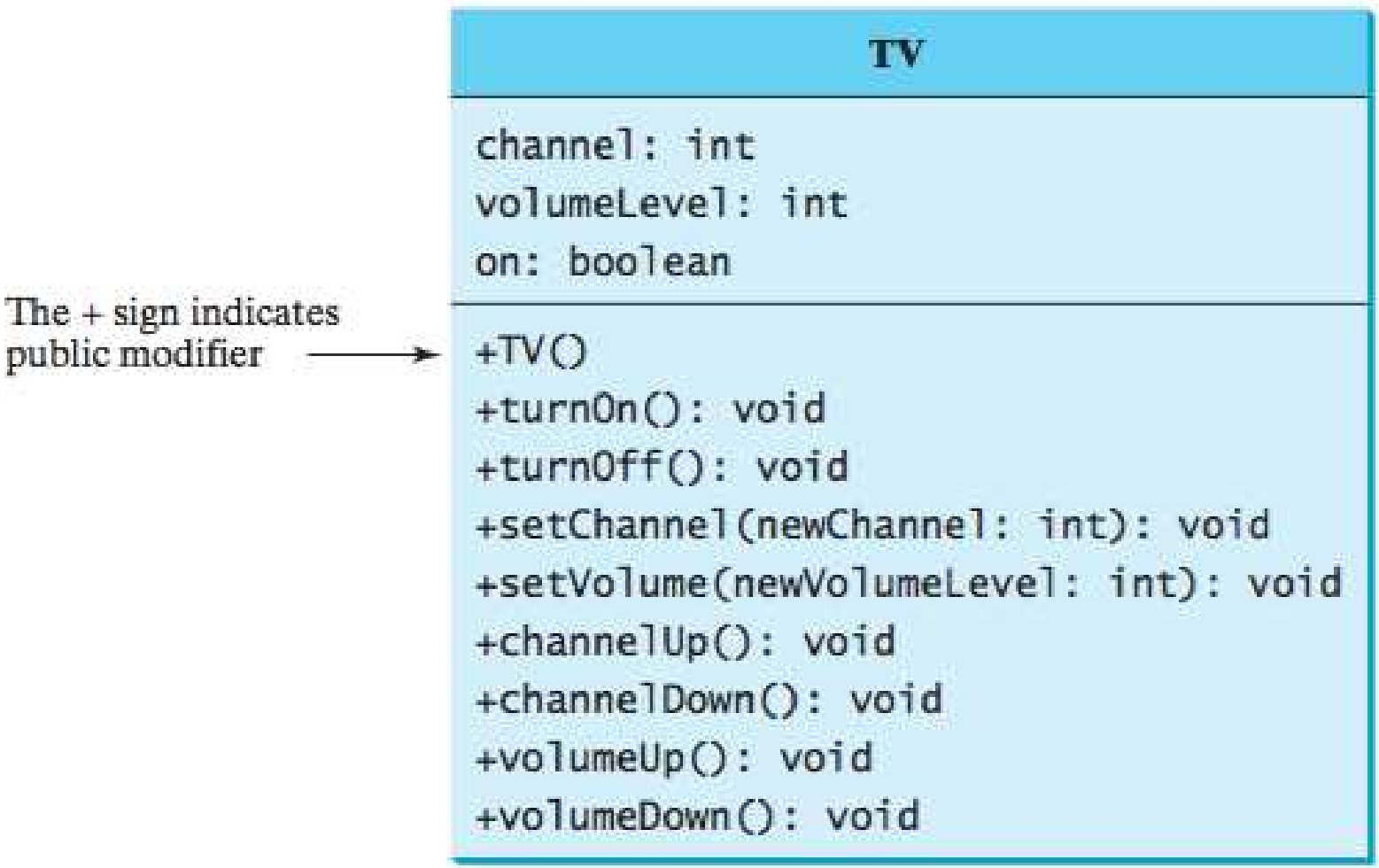


Figure: (Source: (Liang, 2014))

TV Exercise (2/4)

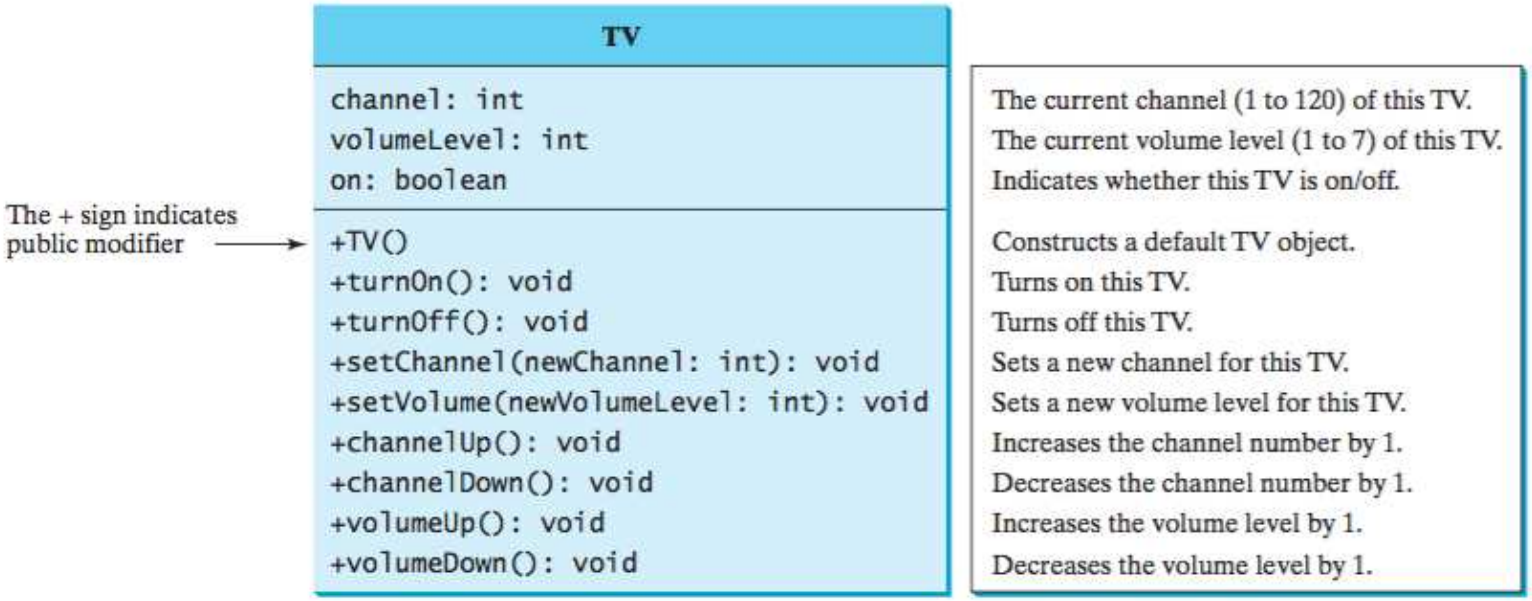


Figure: (Source: (Liang, 2014))

TV Exercise (3/4)

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

    public TV() {}

    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 setVolume(int newVolumeLevel) {
        if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
        volumeLevel = newVolumeLevel;
    }
// ... (continues next slide)
```

TV Exercise (4/4)

```
//  ... (continuation of previous slide)
    public void channelUp() {
        if (on && channel < 120)
        channel++;
    }

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

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

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

In retrospect, based on the concepts seen until now (1/4):

What is an object? Any ideas?

In retrospect, based on the concepts seen until now (2/4):

What is an object? Any ideas?

1. Everything is an object:
   - An object is just a fancy variable:
     - Stores data;
   - Requests can be made to an object;
   - Concepts can be represented as objects;

In retrospect, based on the concepts seen until now (3/4):

What is an object? Any ideas?

2. Program consists of objects telling each other what to:
   - A request is a method call to a particular object;
3. Each object has its own memory made up of other objects:
   - Allows for complex programs to be built...
   - ...while hiding behind the simplicity of objects;

In retrospect, based on the concepts seen until now (4/4):

What is an object? Any ideas?

4. Every object has a type:
   - Each object is an instance of a class;
   - ``Class'' is synonymous with ``type'';
5. All objects of a particular type can receive the same messages:
   - This concept will be extended in further chapters;

Constructing Objects Using Constructors

Once a class is defined:

How can a class be used to construct a new object? Any ideas?

Constructing Objects Using Constructors

Once a class is defined:

How can a class be used to construct a new object? Any ideas?

- By using something called a **constructor**

But what is a constructor? Any ideas?

Constructing Objects Using Constructors

Once a class is defined:

How can a class be used to construct a new object? Any ideas?

- By using something called a **constructor**

But what is a constructor? Any ideas?

- **Constructors** are responsible for initializing objects;

Constructors are a special kind of method (1/2):

- Constructors must have the same name as the class itself;
- Constructors do not have a return type;
- Constructors are invoked using the **new** operator:

```
new ClassName( arguments );
```

Constructors are a special kind of method (2/2):

- Constructors can be **overloaded**:

> What does it mean to overload a method? Any ideas?

Constructors are a special kind of method (2/2):

- Constructors can be **overloaded**:

What does it mean to overload a method? Any ideas?

  - Multiples methods can have same name but different signatures;

```
// Create circle with radius 1
Circle c1 = new Circle( );

// Create circle with radius 25
Circle c2 = new Circle( 25 );
```

Important observation:

- Every class has a **default** constructor:
  - With an empty body;
  - With no arguments;

Accessing objects via Reference Variable

Newly created objects are created in memory, but...

How can we access an object? Any ideas?

How can we access the attributes of an object? Any ideas?

How can we access the methods of an object? Any ideas?

Lets have a look at each one of these questions individually =)

Reference variables and reference types

How can we access an object? Any ideas?

Objects are accessed via the object's reference variables:

- Contain **references** to the objects;
- Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

- Examples:

```
Circle c1 = new Circle();
```

But what is a reference? Any ideas?

But what is a reference? Any ideas?

- Objects are stored in memory;
- Memory positions are accessed through addresses;
- **Reference:** Address where an object's variables and methods are stored.
  - Reference can be thought of as **pointers**;
  - **However:** There are no explicit pointers or pointer arithmetic in Java;

Accessing an Object's Data and Methods

How can we access the attributes of an object? Any ideas?

How can we access the methods of an object? Any ideas?

Through the dot operator **(.)**:

- objectRefVar.dataField references a data field in the object, *e.g.:*

```
// Access c1 attribute radius
c1.radius;
```

- objectRefVar.method(arguments) invokes a method on the object, *e.g.:*

```
// Calculate c1 area
c1.getArea()
```

Reference Data Fields

To what values are the attributes of an object initialized? Any ideas?

Reference Data Fields

To what values are the attributes of an object initialized? Any ideas?

Consider the following Student class:

```
class Student {
    String name;
    int age;
    boolean isEnrolled;
    char gender;
}
```

- What is the default value of attribute name?
- What is the default value of attribute age?
- What is the default value of attribute isEnrolled?
- What is the default value of attribute gender?

Reference Data Fields

To what values are the attributes of an object initialized? Any ideas?

Consider the following Student class:

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

- What is the default value of attribute name? **null**
- What is the default value of attribute age? **0**
- What is the default value of attribute isEnrolled? **false**
- What is the default value of attribute gender? **\u0000**

Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value:

- When a variable is declared: compiler knows what type of value to hold;

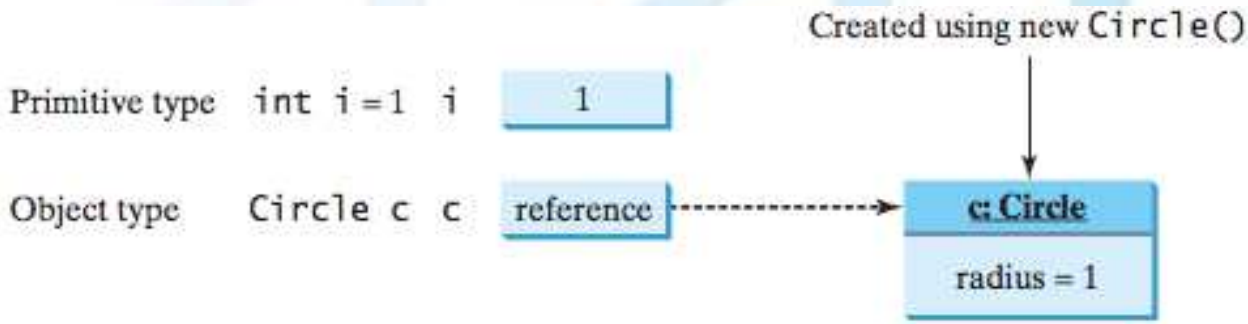


Figure: A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory. (Source: (Liang, 2014))

But what happens when we try to assign one variable to another? Any ideas?

But what happens when we try to assign one variable to another? Any ideas?

Well it **depends**:

- Is the variable primitive?
- Is the variable an object?

Lets see some examples...

Consider the following **primitive** attribution:

```
int i = 1;
int j = 2;

i = j;
```

What is the value of i?

Consider the following **primitive** attribution:

```
int i = 1;
int j = 2;

i = j;
```

What is the value of i?

- $i = 2$

For a variable of **primitive** type:

- the value is copied;

Consider the following **object** attribution:

```
Circle  c1 = new Circle( 5 );
Circle  c2 = new Circle ( 9 );

c1 = c2
```

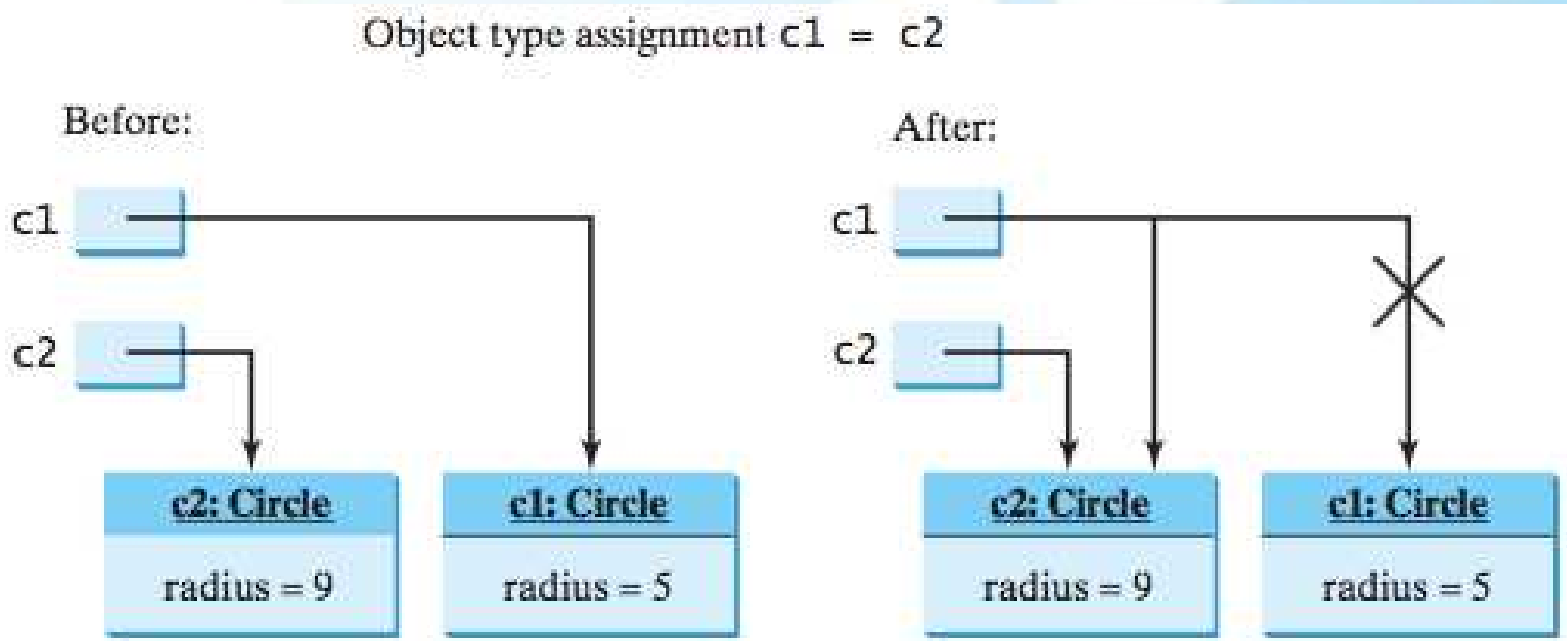
What happens in this case? Any ideas?

Consider the following **object** attribution:

```
Circle  c1 = new Circle( 5 );
Circle  c2 = new Circle ( 9 );

c1 = c2
```

What happens in this case? Any ideas?



Some **important** observations of the previous example:

- After the assignment statement c1 = c2:
  - c1 points to the same object referenced by c2;
  - Object previously referenced by c1 is no longer useful;
  - The object is know as **garbage**;
- Garbage occupies memory space:
  - Java runtime system detects garbage and automatically reclaims the space;
  - This process is called **garbage collection**.

Static Variables, Constants and Methods

Suppose that you create the following objects:

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

- Radius in circle1 is **independent** of the radius in circle2:
  - $\neq$ objects $\rightarrow$ stored in a $\neq$ memory location.
  - *I.e.* Changes made to circle1's radius do not affect circle2's radius, and vice versa;

But what if we need to share memory between $\neq$ objects of the same class? Any ideas?

But what if we need to share memory between $\neq$ objects of the same class? Any ideas?

This can be done through the use of **static** variables:

- Variables are stored in a **common** memory location;
- If one object changes the value of a static variable
  - All objects of the same class are affected.
- Java supports static methods as well as static variables:
  - Static methods can be called without creating an instance of the class.

But why do we need **static** variables? Any ideas?

But why do we need **static** variables? Any ideas?

This is equivalent to the question:

But why do we need **global** variables? Any ideas?

But why do we need **static** variables? Any ideas?

This is equivalent to the question:

But why do we need **global** variables? Any ideas?

- Global variables are used extensively to pass information;
- This can be problematic when dealing with multi-threaded environments:
  - Ask your OS professor about these;
  - Oh wait, I am your OS professor ;)

Lets see an example why static variables are interesting:

```
public class CircleWithStaticMembers {
    /** The radius of the circle */
    double radius;

    /** The number of objects created */
    static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    CircleWithStaticMembers() {
        radius = 1;
        numberOfObjects++;
    }

    /** Construct a circle with a specified radius */
    CircleWithStaticMembers(double newRadius) {
        radius = newRadius;
        numberOfObjects++;
    }

    /** Return numberOfObjects */
    static int getNumberOfObjects(){
        return numberOfObjects;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }
}
```

What is happening in the previous code? Any ideas?

What is happening in the previous code? Any ideas?

- Modifier **static** in the variable or method declaration;
- Static variable numberOfObjects counts number of circle objects created;
- Each time an object is created the numberOfObjects is incremented;
- Static method getNumberOfObjects was also created:
  - Makes it easy to retrieve the numberOfObjects created;

Now in UML:

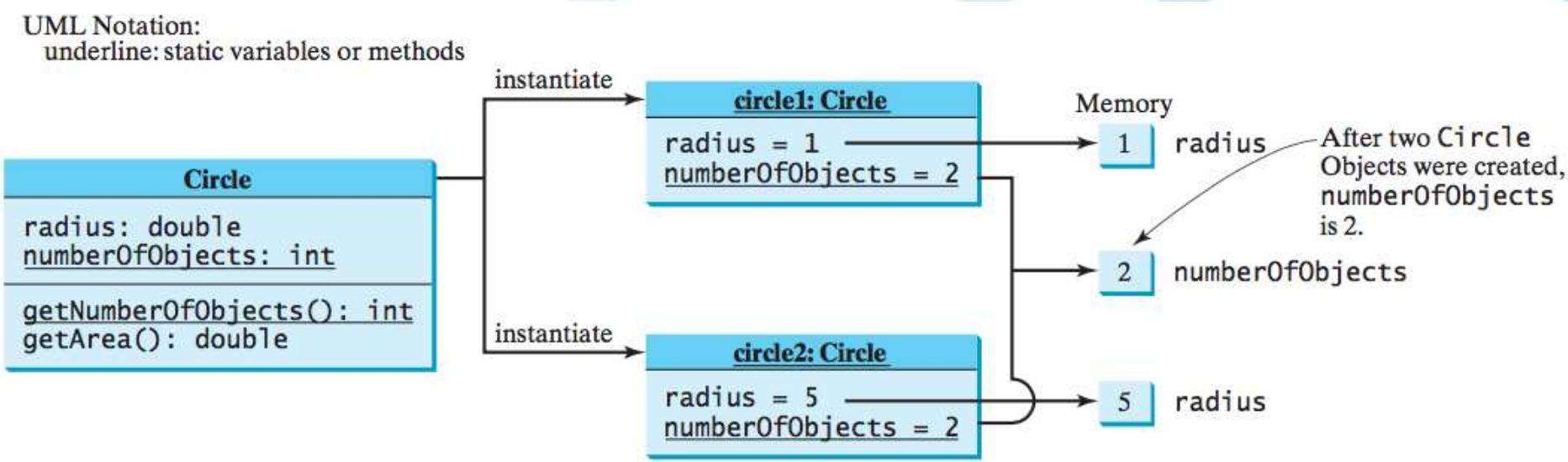


Figure: Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class. (Source: (Liang, 2014))

- Static variables and methods are underlined in the UML class diagram

You did not notice but you were already using **static** variables

Can you tell where in our previous codes we were using **static** variables? Any ideas?

You did not notice but you were already using **static** variables

Can you tell where in our previous codes we were using **static** variables? Any ideas?

Math.PI

- Notice that we are accessing a java class named **Math**;
- But no object of class Math was ever created;
- We are just interested in obtaining the value of a constant named PI:
  - No need for that constant to exist in every object;
  - It would be a waste of memory;

Some important observations:

- An **instance** method can:
  - Invoke an instance or static method;
  - Access an instance or static data field.
- A **static** method can
  - Invoke a static method;
  - Access a static data field;
- However, static method **cannot:**
  - Invoke an instance method;
  - Access an instance data field;

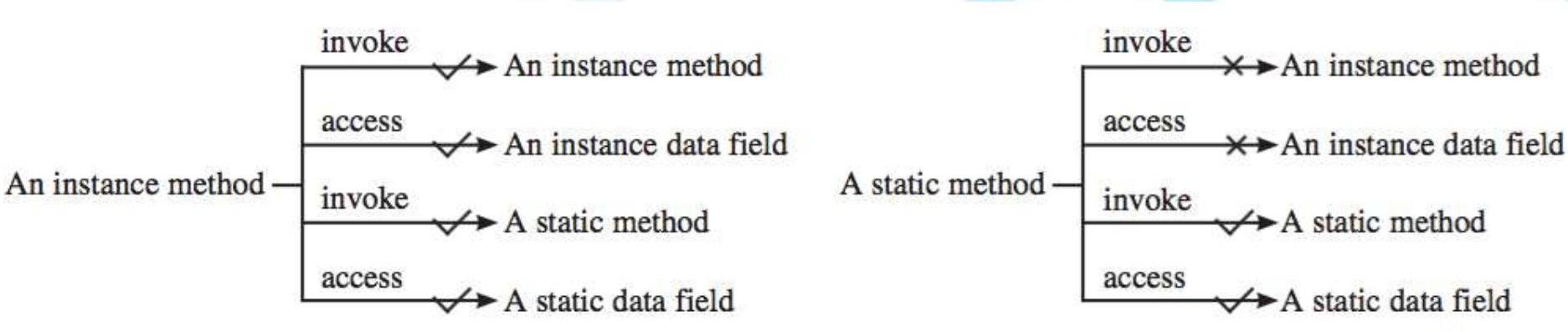


Figure: Difference between instance and static methods (Source: (Liang, 2014))

Why does this happen? Any ideas?

Why does this happen? Any ideas?

- Static methods and static data fields don't belong to a **particular** object;
- They are **global** methods;
- No instance attributes or methods are therefore associated;

Consider the following code:

```
public class A{
    int i = 5;
    static int k = 2;

    public static void main( String [] args ){
        int j = i;
        m1();
    }

    public void m1(){
        i = i + k + m2(i, k);
    }

    public static int m2( int i, int j){
        return (int) (Math.pow(i,j));
    }

}
```

Consider the following code:

```
public class A{
    int i = 5;
    static int k = 2;

    public static void main( String [] args ){
        int j = i;
        m1();
    }

    public void m1(){
        i = i + k + m2(i, k);
    }

    public static int m2( int i, int j){
        return (int) (Math.pow(i,j));
    }

}
```

Can you see anything wrong? Any ideas?

Several **wrong** things:

```
public class A{
    int i = 5;
    static int k = 2;

    public static void main( String [] args ){
        int j = i; // Wrong because i is an instance variable
        m1(); // Wrong because m1() is an instance method
    }

    public void m1(){
        // Correct since instance and static variables and methods
        // can be used in an instance method
        i = i + k + m2(i, k);
    }

    public static int m2( int i, int j ){
        // Correct since pow is accessing arguments i and j
        // from the m2 function, and not the attribute i
        return (int) ( Math.pow( i, j ) );
    }

}
```

So the question now is:

How can we fix the previous code? Any ideas?

So the question now is:

How can we fix the previous code? Any ideas?

```
public class A{
    int i = 5;
    static int k = 2;

    public static void main( String [] args ){
        A a = new A();
        int j = a.i;
        a.m1();
    }

    public void m1(){
        i = i + k + m2(i, k);
    }

    public static int m2( int i, int j){
        return (int) ( Math.pow( i, j ) );
    }

}
```

How do you decide whether a variable or a method should be an instance one or a static one?

How do you decide whether a variable or a method should be an instance one or a static one?

Variable / method **dependent** on a specific instance of the class:

- Should be an **instance** variable or method;

Variable / method **independent** on a specific instance of the class:

- Should be a **static** variable or method. F

Examples

Every circle has its own radius:

- Radius is **dependent** on the specific circle;
- Therefore, radius is an **instance** variable;

getArea() method is dependent on a specific circle:

- Instance method;

Methods Math.random, Math.pow, Math.sin and Math.cos:

- Independent of the instance;
- Therefore, **static** methods;

Visibility Modifiers

Sometimes:

- **Useful** to control access to attributes / methods from other classes:

How can this be done? Any ideas?

Visibility Modifiers

Sometimes:

- **Useful** to control access to attributes / methods from other classes:

How can this be done? Any ideas?

- Using **visibility Modifiers** ;)

Visibility Modifiers

Sometimes:

- **Useful** to control access to attributes / methods from other classes:

How can this be done? Any ideas?

- Using **visibility Modifiers** ;)

So what is a visibility modifier? Any ideas?

Visibility Modifiers

Sometimes:

- **Useful** to control access to attributes / methods from other classes:

How can this be done? Any ideas?

- Using **visibility Modifiers** ;)

So what is a visibility modifier? Any ideas?

So what is a visibility modifier? Any ideas?

Set of **keywords** that:

- Change the **visibility** of classes, methods and attributes:
  - To denote how they can be **accessed** from other classes;
- If no visibility modifier is used then by default:
  - Classes, methods, and data fields are accessible in the same **package**;

But wait:

What is a **package**? Any ideas?

But wait:

What is a **package**? Any ideas?

- Packages can be used to organize classes;
- Requires the following first line:

```
package packageName;
```

- If a class is defined without the **package** statement:
  - ``default'' package is used;

Now back to the modifiers...

What are the modifiers available in Java? Any ideas?

What are the modifiers available in Java? Any ideas?

- **public**
- **private**
- **protected**

Can you guess what each one of the modifiers does? Any ideas?

What are the modifiers available in Java? Any ideas?

- **public:**
  - Classes, methods and attributes **can** be accessed from other classes;
- **private:**
  - Classes, methods and attributes **cannot** be accessed from other classes;
- **protected:**
  - Classes, methods and attributes **can** be accessed from **descendant** classes;

Example

private modifier:

- methods and data fields accessible only from within its own class:

```
package p1;

public class C1{
    public int x;
    int y;
    private int z;

    public void m1(){}

    void m2(){}

    private void m3(){}
}
```

```
package p1;

public class C2{
    void aMethod(){
        C1 o = new C1();
        //Can access o.x
        //Can access o.y
        //Cannot access o.z
        //Can invoke o.m1()
        //Can invoke o.m2()
        //Cannot invoke o.m3()
    }
}
```

```
package p2;

public class C3{
    void aMethod(){
        C1 o = new C1();
        //Can access o.x
        //Cannot access o.y
        //Cannot access o.z
        //Can invoke o.m1()
        //Cannot invoke o.m2()
        //Cannot invoke o.m3()
    }
}
```

Example

If a class is not defined as public:

- Can be accessed only within the same package;

```
package p1;

class C1{
    ...

}
```

```
package p1;

public class C2{
    // can access C1

}
```

```
package p2;

public class C3{
    // Cannot access C1
    // Can access C2
}
```

Visibility modifier specifies how:

- Attributes / methods in a class can be accessed from outside the class;
- No restriction exists on accessing attributes / methods from inside the class;
- Example:

```
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;
    }
}
```

Listing 1: Correct: Object c of class C can access its private members;

But what about the following code? Any ideas?

```
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;
    }
}
```

Listing 2: Correct: Object c of class C can access its private members;

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


}
```

Listing 3: Correct/Wrong?

But what about the following code? Any ideas?

```
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;
    }
}
```

Listing 4: Correct: Object c of class C can access its private members;

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

}
```

Listing 5: Wrong: x and convert are private in class C. Compile error.

Data Field Encapsulation

Recall our previous example: CircleWithStaticMembers

- Attributes **radius** and **numberOfObjects** can be modified directly;

Is this a good practice? Any ideas?

Data Field Encapsulation

Recall our previous example: CircleWithStaticMembers

- Attributes **radius** and **numberOfObjects** can be modified directly;

Is this a good practice? Any ideas?

Not a good practice, for two reasons:

- Data tampering: attribute may be wrongly set;
- Class becomes difficult to maintain and vulnerable to bugs;

What does it mean: ``Class becomes difficult to maintain and vulnerable to bugs''? Any ideas?

What does it mean: ``Class becomes difficult to maintain and vulnerable to bugs''? Any ideas?

Suppose we want to modify CircleWithStaticMembers class:

- To ensure that the radius is nonnegative:
  - After other programs have already used the class;
- Modifications to be done:
  - Update CircleWithStaticMembers;
  - Update programs that use class CircleWithStaticMembers:
    - These may have modified radius directly (*e.g.,* c1.radius = -5).

Better to have an alternative approach...

To prevent direct modifications of attributes:

- Declare the attributes as **private**, a.k.a. **encapsulation:**
  - Private attributes cannot be accessed by an object from outside the class;

However, an object often needs to retrieve and modify a data field:

How can we perform such accesses / modifications? Any ideas?

However, an object often needs to retrieve and modify a data field:

How can we perform such accesses / modifications? Any ideas?

To make a private data field **accessible**:

- Provide a getter method to return its value;
- **Getter** methods have the following signature:

```
public returnType getPropertyName()
```

- If the returnType is boolean, getter is be defined as follows:

```
public boolean isPropertyName()
```

However, an object often needs to retrieve and modify a data field:

How can we perform such accesses / modifications? Any ideas?

To enable a private data field to be **modified**

- Provide a setter method to set a new value;
- **Setter** methods have the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Example

Create a new circle class with:

- Private data-field radius;
- Associated getters and setters;

Create a new circle class (UML + Java) with:

- Private data-field radius;
- Associated getters and setters;

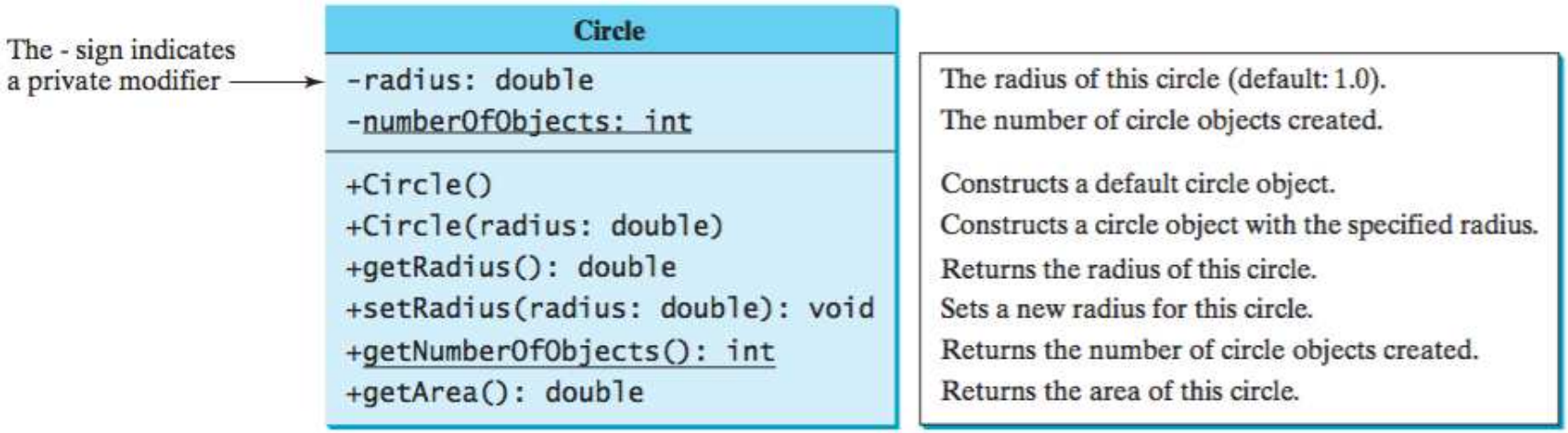


Figure: Circle class encapsulates circle properties and provides getter/setter and and other methods (Source: (Liang, 2014))

```
public class CircleWithPrivateDataFields {

    /** The radius of the circle */
    private double radius = 1;

    /** The number of objects created */
    private static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    public CircleWithPrivateDataFields () {
        numberOfObjects++;
    }

    /** Construct a circle with a specified radius */
    public CircleWithPrivateDataFields (double newRadius) {
        radius = newRadius;
        numberOfObjects++;
    }

    /** Return radius */
    public double getRadius() {
        return radius;
    }

    ...
}
```

```
...

    /** Set a new radius */
    public void setRadius(double newRadius) {
        radius = (newRadius >= 0) ? newRadius : 0;
    }

    /** Return numberOfObjects */
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }

    /** Return the area of this circle */
    public double getArea() {
        return radius * radius * Math.PI;
    }

}
```

Passing Objects to Methods

Objects can be passed to methods:

- *I.e.* the reference of the object is actually passed;
- Example:

```
public static void main(String () args) {
    CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(5.0);
    printCircle (myCircle);
}

public static void printCircle ( CircleWithPrivateDataFields c ){
    System.out. println ("The area of the circle of radius "
    + c.getRadius() + " is " + c.getArea());
}
```

Java uses pass-by-value:

- Value of myCircle is passed to the printCircle method;
- Value is a reference to a Circle object.

What is the output of the following code?

```
public class TestPassObject {
    public static void main(String[] args) {
        CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(1);
    }

        // Print areas for radius 1, 2, 3, 4, and 5.
        int n = 5;
        printAreas(myCircle, n);

    // See myCircle.radius and times
    System.out.println("\n" + "Radius is " + myCircle.getRadius());
    System.out.println("n is " + n);

    /** Print a table of areas for radius */
    public static void printAreas( CircleWithPrivateDataFields c, int times) {
        System.out.println("Radius \t\tArea");
        while (times >= 1) {
            System.out.println(c.getRadius() + "\t\t" + c.getArea());
            c.setRadius(c.getRadius() + 1);
            times--;
        }
    }
}
```

Output:

```
Radius     Area
1.0        3.141592653589793
2.0        12.566370614359172
3.0        29.274333882308138
4.0        50.26548245743669
5.0        79.53981633974483
Radius is 6.0
n is 5
```

Array of Objects

Arrays of objects can also be created, example:

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

Does this mean that 10 circleArray objects have been created? Any ideas?

Array of Objects

Arrays of objects can also be created, example:

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

Does this mean that 10 circleArray objects have been created? Any ideas?

- No....
- Only space in memory for 10 circleArray's has been created;

So what is missing? Any ideas?

So what is missing? Any ideas?

Initialization...

How can the array be initialized then? Any ideas?

So what is missing? Any ideas?

Initialization...

How can the array be initialized then? Any ideas?

```
for (int i = 0; i < circleArray.length; i++) {
    circleArray(i) = new Circle();
}
```

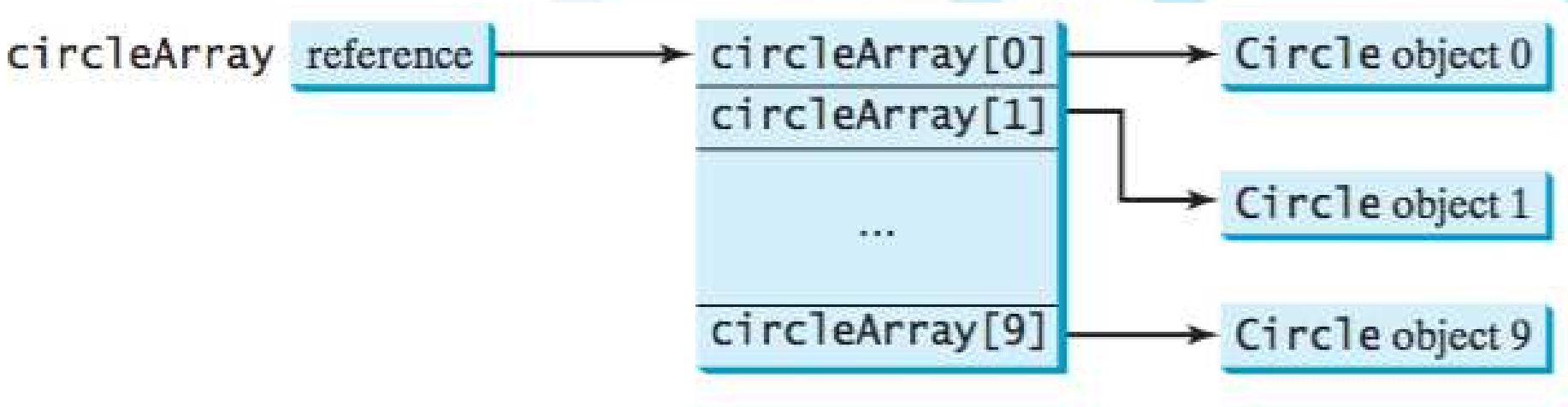


Figure: In an array of objects, an element of the array contains a reference to an object. (Source: (Liang, 2014))

Immutable Objects and Classes

Normally:

- Objects are created and its contents can later be changed;

However, sometimes it is desirable to:

- Create an object whose contents cannot be changed once:
  - The object has been created.
- These are known as **immutable objects / classes**;

How can we specify immutable objects / classes? Any ideas?

How can we specify immutable objects / classes? Any ideas?

If a class is **immutable**, then:

- All **attributes** are **private**;
- Cannot contain public setter methods for any attributes;

However:

- Class with private attributes and no mutators is not necessarily immutable.

Lets see an example...

Example

```
public class Student {
    private int id;
    private String name;
    private java.util.Date dateCreated;

    public Student(int ssn, String newName) {
        id = ssn;
        name = newName;
        dateCreated = new java.util.Date();
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public java.util.Date getDateCreated() {
        return dateCreated;
    }
```

Is this class immutable? Any ideas?

Is this class immutable? Any ideas?

No...:

- Attribute dateCreated is returned using the getDateCreated();
- This is a reference to a Date object:
  - Through this reference, the content for dateCreated can be changed.

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, "John");
        java.util.Date dateCreated = student.getDateCreated();
        dateCreated.setTime(200000); // Now dateCreated field is changed!
        }
}
```

So then what are the requirements for an immutable object / class? Any ideas?

So then what are the requirements for an immutable object / class? Any ideas?

- All data fields must be private;
- There can't be any setter methods;
- No getter methods can return a reference to a mutable attribute;

Scope of Variables

Lets start with something basic:

What is a local variable? Any ideas?

Scope of Variables

Lets start with something basic:

What is a local variable? Any ideas?

- Local variables are declared and used inside a method locally.

But what about the scope of variable within a class?

Lets have a look into the scope rules of all the variables of a class...

Instance and static variables in a class are referred to as:

- Class's variables or attributes;

A variable defined inside a method is referred to as:

- Local variable.

What is the scope of a class variable? Any ideas?

What is the scope of a class variable? Any ideas?

- Entire class: regardless of where the variables are declared;
- Class's variables and methods can appear in any order in the class:
  - Exception: when attribute is initialized based on another attribute;

```
public class Circle {

    public double findArea() {
        return radius * radius * Math.PI;
    }

    private double radius = 1;
}
```

Listing 6: The variable radius and method findArea() can be declared in any order.

```
public class F {
    private int i;
    private int j = i + 1;


}
```

Listing 7: i has to be declared before j because j's initial value is dependent on i.

- Convention: declares attributes at the beginning of the class.

What about local variables? Any ideas?

What about local variables? Any ideas?

If a local variable has the same name as a class's variable:

- Local variable takes precedence and:
  - Class's variable with the same name is hidden

Lets look at an example...

Example (1/3)

```
public class F {
    private int x = 0; // Instance variable
    private int y = 0;

    public F() { }
    public void p() {
        int x = 1; // Local variable
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

What is the output for f.p(), where f is an instance of F? Any ideas?

Example (2/3)

What is the output for f.p(), where f is an instance of F? Any ideas?

Output for f.p() is 1 for x and 0 for y. Here is why:

- x:
  - Declared as a attribute with initial value of 0;
  - Also declared in the method p() with an initial value of 1;
  - Local variable takes precedence over class variable;
    - Therefore x = 1;

Example (3/3)

What is the output for f.p(), where f is an instance of F? Any ideas?

Output for f.p() is 1 for x and 0 for y. Here is why:

- y:
  - Declared outside the method p(), but y is accessible inside the method;
  - Therefore y = 0;

The **this** Reference

this keyword:

- Sometimes: useful for an object to refer to itself:
- Can be used to reference the object's instance members

Lets see an example:

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

    public String toString() {
        return "radius: " + this.radius
        + "area: " + this.getArea(); }
}
```

Listing 8: Equivalent to the code on the right.

```
public class Circle { private double radius;
    ...
    public double getArea() {
        return radius * radius * Math.PI; }

    public String toString() {
        return "radius: " + radius
        + "area: " + getArea(); }
}
```

Listing 9: Equivalent to the code on the left.

However: this keyword is needed to:

- Reference hidden data fields or
- Invoke an overloaded constructor.

How can we then use **this** to reference hidden data fields? Any ideas?

How can we then use **this** to invoke a constructor? Any ideas?

Lets have a look at each one of these questions

Using this to Reference Hidden Data Fields

How can we then use **this** to reference hidden data fields? Any ideas?

- **this** keyword can be used to reference a class's hidden data fields.

Is the following code correct? Any ideas?

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

    public void setI(int i) {
        i = i;
    }

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

    // Other methods omitted
}
```

Example

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

    public void setI(int i) {
        i = i;
    }

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

    // Other methods omitted
}
```

Suppose that f1 and f2 are two objects of F. What does **this** refer to?:

- When f1.setI(10) is executed?
- When f2.setI(45) is executed?
- When F.setK(33) is executed?

Example

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

    public void setI(int i) {
        i = i;
    }

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

    // Other methods omitted
}
```

Suppose that f1 and f2 are two objects of F. What does **this** refer to?:

- When f1.setI(10) is executed?
  - this.i = 10, where **this** refers f1
- When f2.setI(45) is executed?
  - this.i = 45, where **this** refers f2
- When F.setK(33) is executed?
  - F.k = 33. setK is a static method

From the **previous** example:

- Attribute name is used as the parameter in a setter for the data-field;
- In this case: data-field is **hidden** in the setter method:
  - Need to reference the hidden data-field name in the method...
  - ...in order to set a new value to it

Using **this** to Invoke a Constructor

How can we then use **this** to invoke a constructor? Any ideas?

Using **this** to Invoke a Constructor

How can we then use **this** to invoke a constructor? Any ideas?

By following exactly the previous logic:

```
public class Circle {

    private double radius;

    // The this keyword is used to reference the
    // hidden data field radius of the object being constructed
    public Circle(double radius) {
        this.radius = radius;
    }

    // The this keyword is used to invoke another constructor.
    public Circle() {
        this(1.0);
    }
 ...
}
```

Line this(1.0) in 2^{nd} constructor invokes 1^{st} constructor with a double

Important note: Java requires that the

- **this(arg-list)** statement appear 1^{st} in the constructor:
  - before any other executable statements.

References I

Liang, Y. (2014).

Introduction to Java Programming.

Pearson Education.