CANADIAN UNIVERSITY OF BANGLADESH

Inspiring Applied Knowledge

School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-7: **Objects & Classes (Part I)**

Prerequisite: CSE 1101

Semester: Summer 2024

Object-Oriented Problem Solving

Objects & Classes (Part I)

Based on Chapter 9 of “Introduction to Java Programming” by Y. Daniel Liang.

Outline

- Defining Classes for Objects (9.2)
- Declaring and Creating Objects Reference Variables(9.5.1)
- Accessing an object’s members (9.5.2)
- Example: TestCircle.java.
- Constructing objects using constructors (9.4)
- Reference data fields and the null value (9.5.3)
- Difference between variables of primitive types and reference types. (9.5.4)
- UML class diagrams.

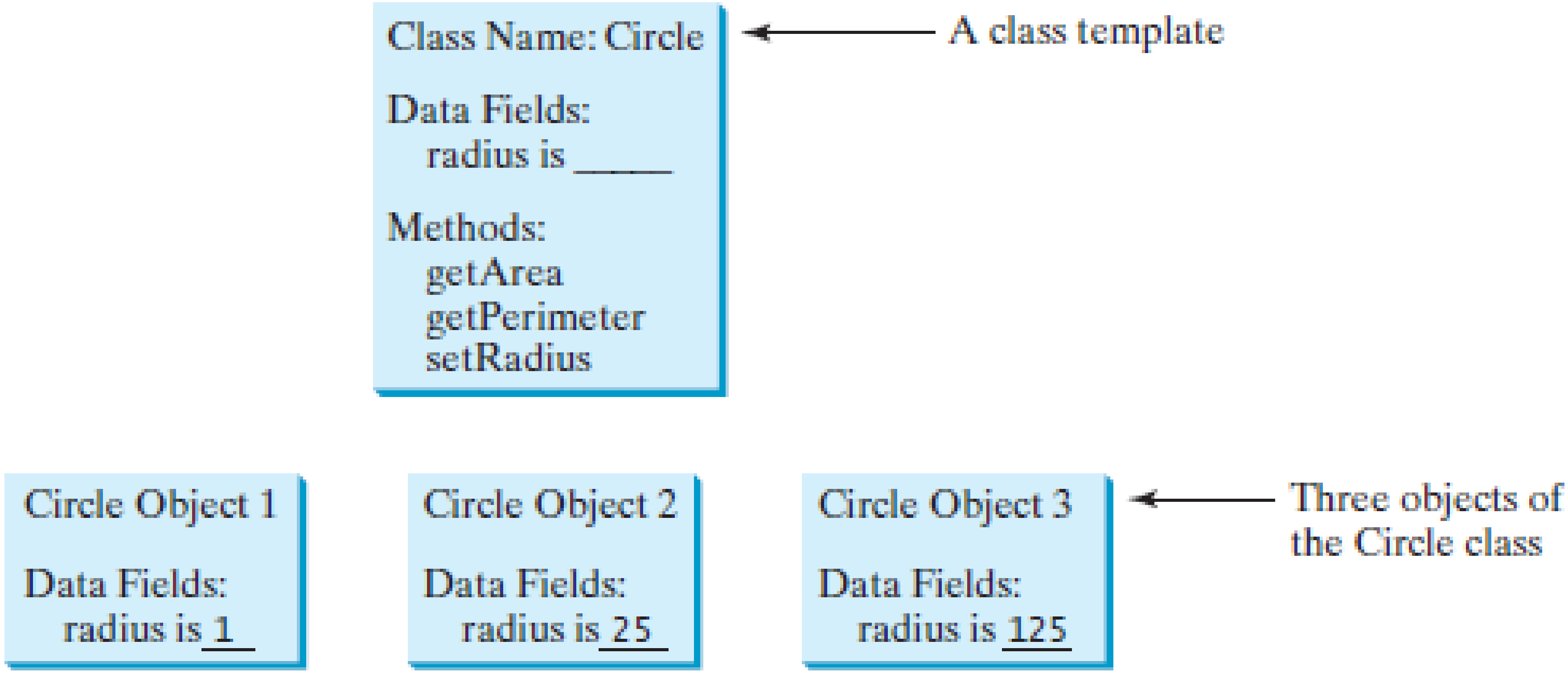
Defining Classes for Objects
What is an Object?

- *Object-oriented programming (OOP)* involves programming using *objects*.
- An *object* represents an entity that can be distinctly identified.
- An object has a unique:
  - *Identity*
  - *State*
    - Also known as its properties or attributes.
    - Represented by data fields with their current values.
  - *Behavior*
    - Also known as its actions.
    - Defined by methods: to invoke a method on an object is to ask the object to perform an action.

Defining Classes for Objects
What is a Class?

- A *class* is a *template*, *blue-print*, or *contract* that defines what an objects data fields and methods will be.
- An object is an instance of a class.
  - You can create many instances of a class.
  - Creating an instance is referred to as *instantiation*.
  - The terms *object* and *instance* are often interchangeable.
- Objects of the same type are defined using a common class.
- A *Java class* uses:
  - Variables to define data fields, and
  - Methods to define actions.

Defining Classes for Objects



Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea
getPerimeter
setRadius

A class template

Circle Object 1

Data Fields:
radius is 1

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of
the Circle class

Defining Classes for Objects Example: Circle Class

```
class Circle{
    double radius;
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Class Circle has one variable of type double called radius.

Class Circle has one void method called setRadius which takes one double parameter and assigns it to the variable radius.

- The **Circle** class is different from all of the other classes you have seen thus far.
  - It does not have a **main** method and therefore cannot be run; it is merely a definition for circle objects.

Declaring and Creating Objects Reference Variables

- A class is essentially a *programmer-defined* type.
- Objects are accessed via the object's *reference variables*, which contain references to the objects.
- The syntax to *declare* an object reference variable is:

  ClassName objectRefVar;
- Example:

  Circle myCircle;
- A class is a *reference type*: a variable of the class type can reference an instance of the class.

Declaring and Creating Objects Reference Variables (Cont.)

- To *create* an object and assign its reference to a declared object reference variable:

  objectRefVar = new ClassName ();

- Example:

  myCircle = new Circle();

- The variable *myCircle* holds a reference to a *Circle* object.
  - An object reference variable that appears to hold an object actually contains a reference to that object.

Declaring and Creating Objects Reference Variables (Cont.)

- A single statement can be used to combine

1) the declaration of an object reference variable,
2) the creation of an object, and
3) the assigning of an object reference to the variable as follows:

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

- Example:

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

Accessing an Object’s Members

- In OOP, object’s members are its *data fields* and *methods*.
- An object’s data can be accessed and its methods invoked using the *dot operator (.)*.
  - Also known as the *object member access operator*:
- To reference a data field in an object:
  - *objectRefVar.dataField*
- Example:
  - *myCircle.radius*
- To invoke a method on an object:
  - *objectRefVar.method(arguments)*
- Example:
  - *myCircle.setRadius(5);*

```
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;
}
}
```

The **Circle** class is different from all of the other classes you have seen thus far. It does not have a **main** method and therefore cannot be run; it is merely a definition for circle objects.

The class that contains the **main** method will be referred to in this book, for convenience, as the *main class*.

The illustration of class templates and objects in Figure 9.2 can be standardized using *Unified Modeling Language (UML)* notation. This notation, as shown in Figure 9.4, is called a *UML class diagram*, or simply a *class diagram*.

In the class diagram, the data field is denoted as dataFieldName: dataFieldType The constructor is denoted as ClassName(parameterName: parameterType)

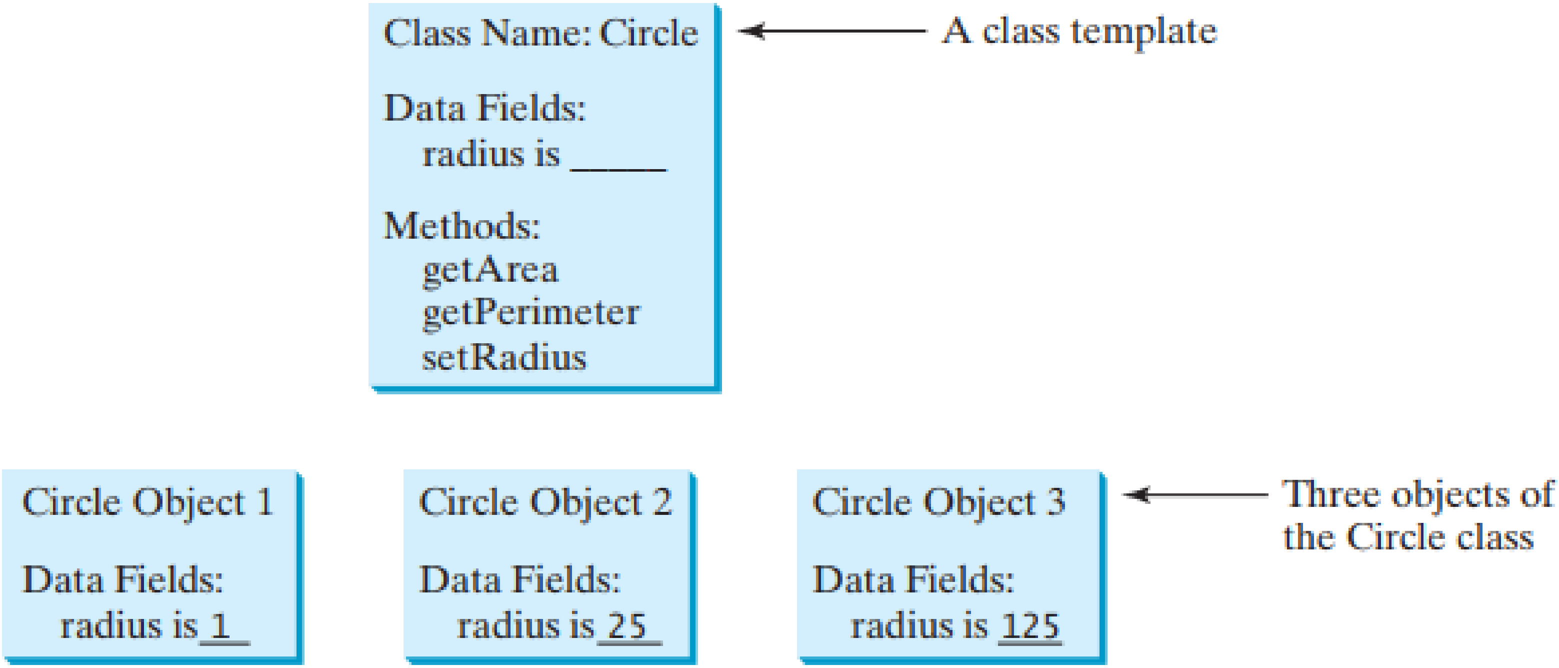


FIGURE 9.2 A class is a template for creating objects.

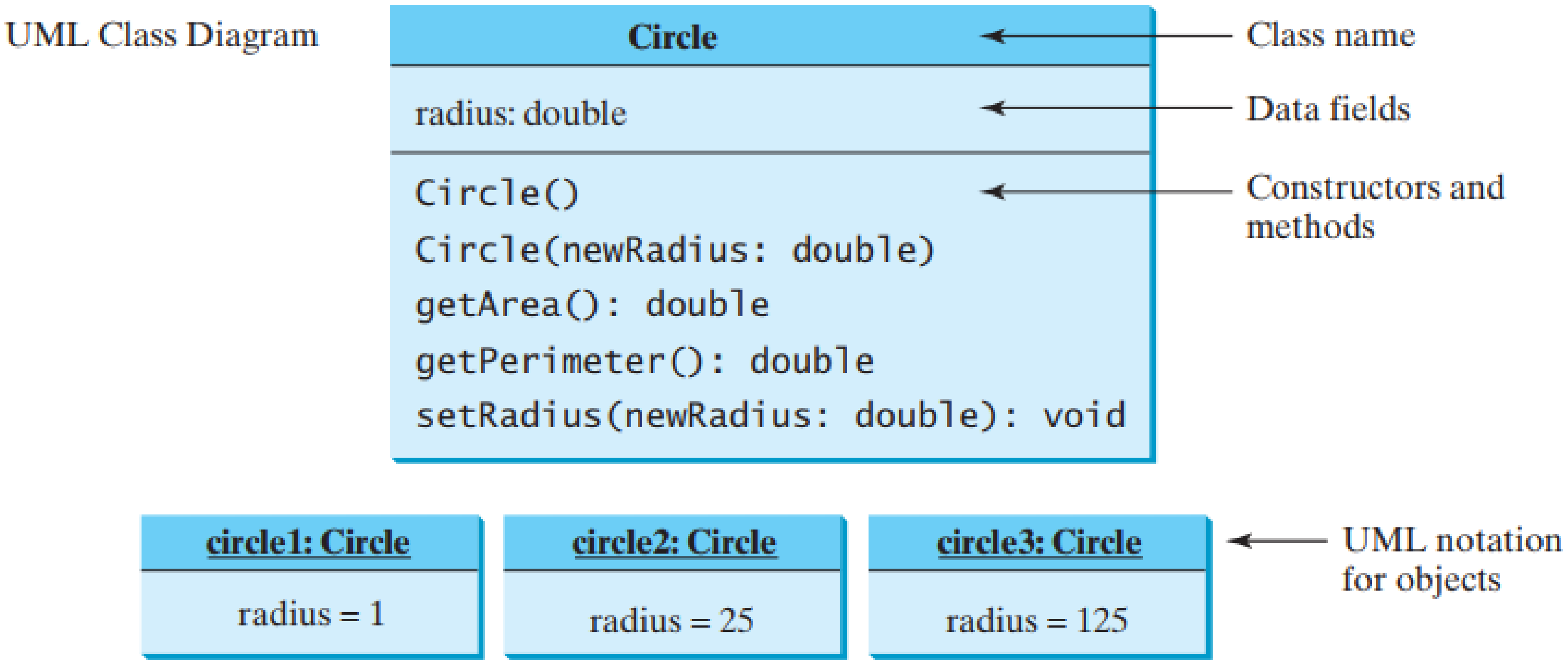


FIGURE 9.4 Classes and objects can be represented using UML notation.

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

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

 // Create a circle with radius 125
SimpleCircle circle3 = new SimpleCircle(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());
}
}
// Define the circle class with two constructors
class SimpleCircle {
double radius;

/** Construct a circle with radius 1 */
SimpleCircle() {
radius = 1;
 }
```

```
/** Construct a circle with a specified radius */
SimpleCircle(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 a new radius for this circle */
void setRadius(double newRadius) {
radius = newRadius;
}
}
```

Following are the several simple and understandable examples in Java that explain how to define classes and create objects.

// Example 1: Basic Class Definition and Object Creation

```
// Define a simple class named 'Car'
class Car {
    // Attributes (fields)
    String model;
    int year;
    // Method to display car details
    void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create an object of the 'Car' class
        Car myCar = new Car();

        // Set the attributes of the object
        myCar.model = "Toyota Camry";
        myCar.year = 2020;

        // Call the method to display the car's details
        myCar.displayInfo();
    }
}
```

Explanation:

- A “Car” class is defined with two attributes: “model” and “year”.
- A method “displayInfo” is created to display the car's details.
- In the “main” method, an object of the “Car” class is created, and its attributes are set. The “displayInfo” method is then called to print the details.

// Example 2: Class with Constructor

```
// Define a class named 'Student'
class Student {
    // Attributes (fields)
    String name;
    int age;
    // Constructor to initialize the attributes
    Student(String n, int a) {
        name = n;
        age = a;
    }
// Method to display student details
    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
public class Main {
    public static void main(String[] args) {
        // Create objects of the 'Student' class using the constructor
        Student student1 = new Student("Alice", 20);
        Student student2 = new Student("Bob", 22);

        // Call the method to display the student's details
        student1.displayInfo();
        student2.displayInfo();
    }
}
```

Explanation:

- The “Student” class has a constructor to initialize the attributes “name” and “age”.

- Objects of the “Student” class are created using the constructor, and the “displayInfo” method is called to print the details.

```
// Example 3: Class with Methods
// Define a class named 'Rectangle'
class Rectangle {
    // Attributes (fields)
    double length;
    double width;
    // Method to calculate area
    double calculateArea() {
        return length * width;
    }
// Method to calculate perimeter
    double calculatePerimeter() {
        return 2 * (length + width);
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an object of the 'Rectangle' class
        Rectangle rect = new Rectangle();
        // Set the attributes
        rect.length = 5.0;
        rect.width = 3.0;
        // Call the methods and display the results
        System.out.println("Area: " + rect.calculateArea());
        System.out.println("Perimeter: " + rect.calculatePerimeter());
    }
}
```

Explanation:

- The “Rectangle” class has methods to calculate the area and perimeter.

- An object of the “Rectangle” class is created, and its attributes are set. The methods are then called to calculate and display the area and perimeter.

```
// Example 4: Class with Multiple Constructors

// Define a class named 'Book'
class Book {
    // Attributes (fields)
    String title;
    String author;
    double price;
    // Constructor 1: No parameters
    Book() {
        title = "Unknown";
        author = "Unknown";
        price = 0.0;
    }
    // Constructor 2: Parameters for all fields
    Book(String t, String a, double p) {
        title = t;
        author = a;
        price = p;
    }
// Method to display book details
    void displayInfo() {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Price: $" + price);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create objects using different constructors
        Book defaultBook = new Book();
        Book specificBook = new Book("Java Programming",
"John Doe", 29.99);
        // Display details of the books
        defaultBook.displayInfo();
        specificBook.displayInfo();
    }
}
```

Explanation:

- The “Book” class has two constructors: one without parameters (default values) and another with parameters to initialize all attributes.

- Two objects are created using different constructors, and their details are displayed using the “displayInfo” method.

Example: TestCircle.java

```
public class TestCircle{
    public static void main (String [] args){
            Circle circle1 = new Circle ();
            circle1.setRadius(5);
            System.out.println(“The radius of circle-1 is ”+circle1.radius);
            Circle circle2 = new Circle();
            circle2.radius = 1;
            System.out.println(“The area of circle-2 is ”+circle2.getArea());
    }
}
class Circle{
    double radius;
    void setRadius (double newRadius){
            radius = newRadius;
    }
   double getArea(){
            return radius*radius*22.0/7.0;
    }
}
```

The public class must have the same name as the file name.

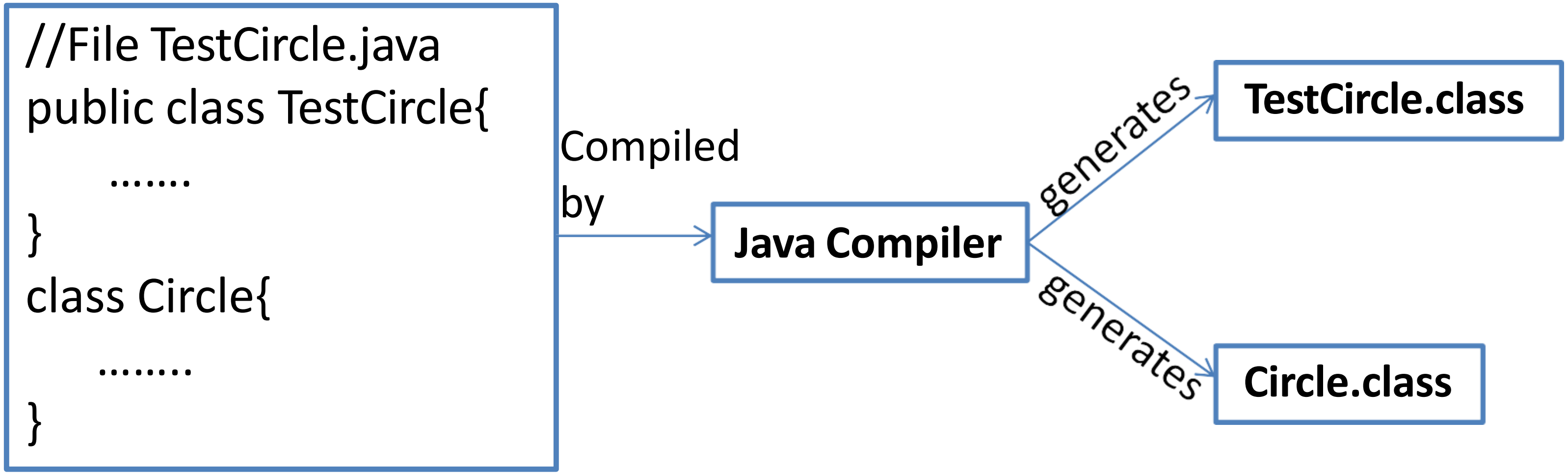
Only one class in a file can be a public class.

Example: TestCircle.java (Cont.)

```
public class TestCircle{
    public static void main (String [] args){
        Circle circle1 = new Circle ();
        circle1.setRadius(5);
        System.out.println(“The radius of circle-1 is ”+circle1.radius);
        Circle circle2 = new Circle();
        circle2.radius = 1;
        System.out.println(“The area of circle-2 is ”+circle2.getArea());
    }
}
class Circle{
    double radius;
    void setRadius (double newRadius){
        radius = newRadius;
    }
    double getArea(){
        return radius*radius*22.0/7.0;
    }
}
```

Remember: this is where the program starts execution.

Example: TestCircle.java (Cont.)



```
//File TestCircle.java
public class TestCircle{
      .......
}
class Circle{
     ........
}
```

Compiled by

Java Compiler

generates

TestCircle.class

generates

Circle.class

Constructing Objects Using Constructors

- A *constructor* is invoked to create an object using the *new* operator.
- *Constructors* are a special kind of method.
- They have three peculiarities:
  - A constructor 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.
    - They play the role of initializing objects.

Constructing Objects Using Constructors (Cont.)

- A class may be defined without constructors.
- In this case, a *default constructor* is provided automatically:
  - A *default constructor* is a *public no-argument* constructor with an empty body which is implicitly defined in the class.
  - A *default constructor* is provided only if there are no other constructors explicitly defined in the class.

Constructing Objects Using Constructors (Cont.)

- The constructor has exactly the same name as its defining class.
- To construct an object from a class, invoke a constructor of the class using the *new* operator, as follows:
  - *new ClassName(arguments);*
- Like regular methods, constructors can be overloaded.
  - Multiple constructors can have the same name but different signatures.
  - Makes it easy to construct objects with different initial data values.

Example TestCircle.java Revisited

```
public class TestCircle{
    public static void main (String [] args){
        Circle circle1 = new Circle (5);
        System.out.println(“The radius of this circle is   “+circle1.radius);
    }
}
class Circle{
    double radius;
    Circle (double initialRadius){
        radius = initialRadius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

Constructors Overloading

```
class Circle{
    double radius;
    Circle(){
        radius = 1;
    }
    Circle (double initialRadius){
        radius = initial Radius;
    }
    void setRadius (double newRadius){
        radius = newRadius;
    }
}
```

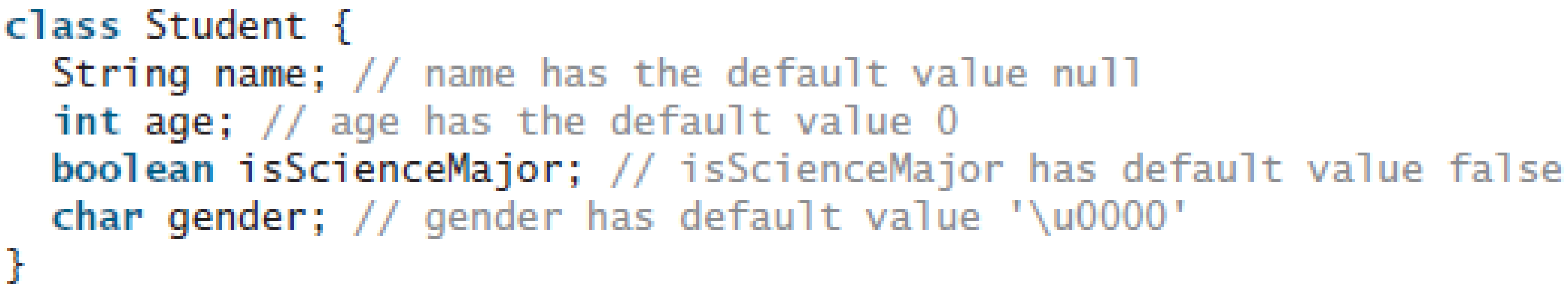
Circle myFirstCircle = new Circle ();

Circle mySecondCircle = new Circle(5);

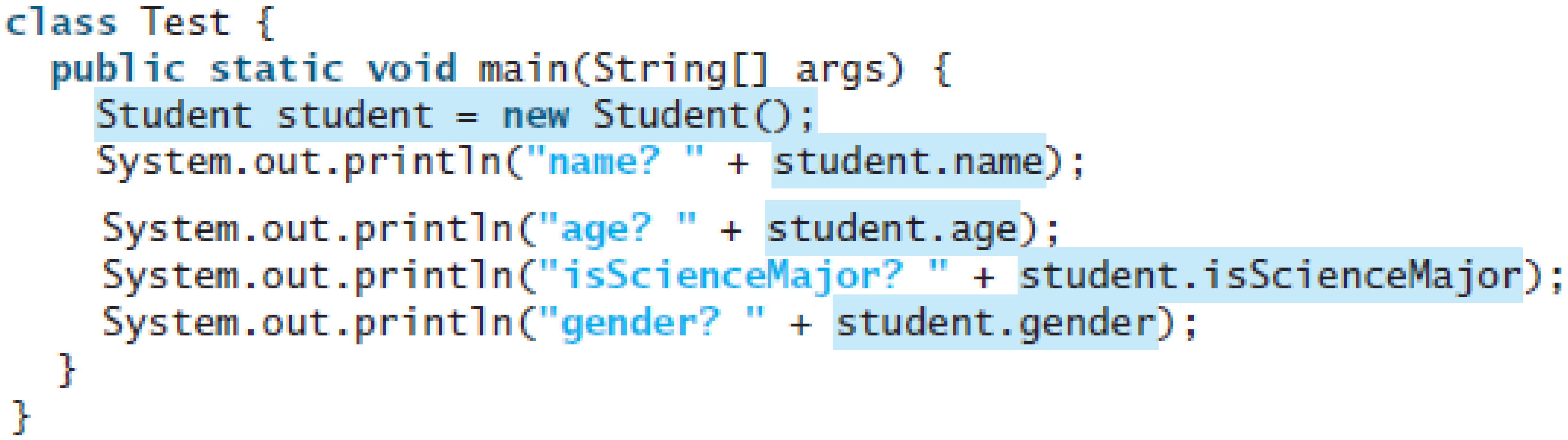
Reference Data Fields and the *null* Value

- Java assigns default values to data fields when an object is created.
  - *0* for *numeric* type.
  - *false* for a *boolean* type.
  - *\u0000* for a *char* type.
  - *Null* for a *reference* type.
    - *Null* is a special literal used for reference types.
- *NullPointerException* is a common runtime error. It occurs when you invoke a method on a reference variable with a *null* value.
- However, Java assigns no default value to a local variable inside a method.

Reference Data Fields and the *null* Value (Cont.)



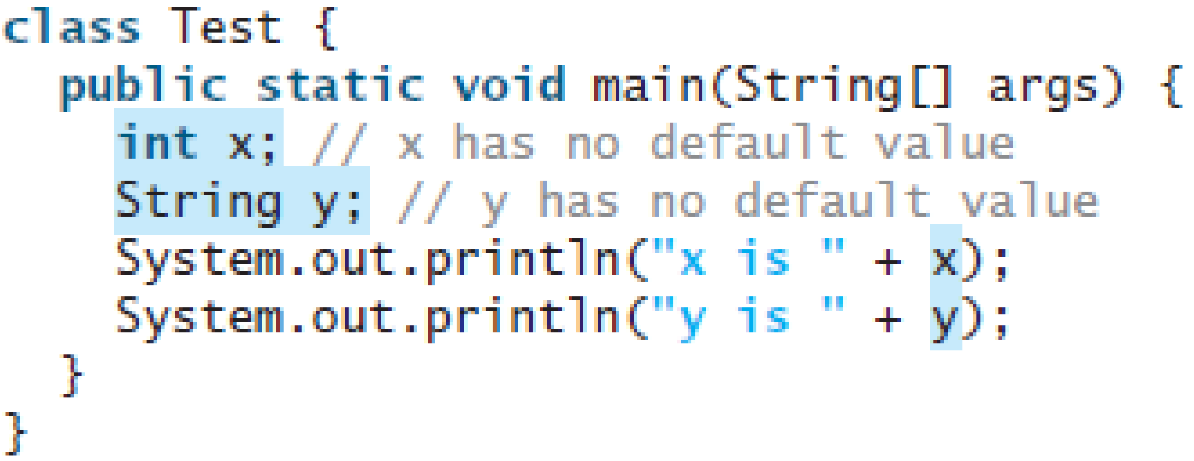
```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```



```
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);
  }
}
```

Reference Data Fields and the *null* Value

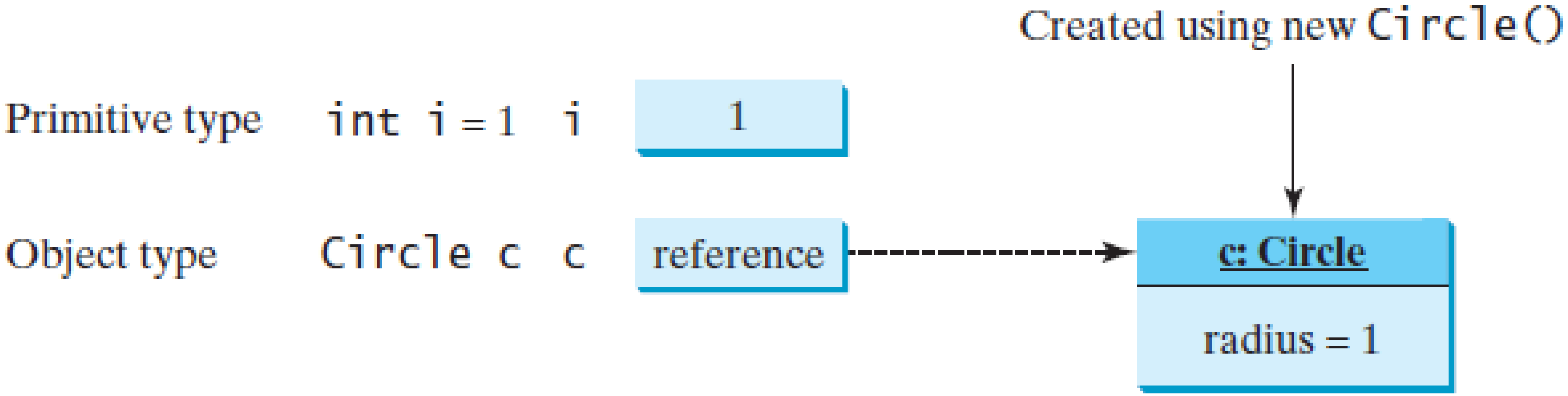
- The following code has a compile error, because the local variables *x* and *y* are not initialized:



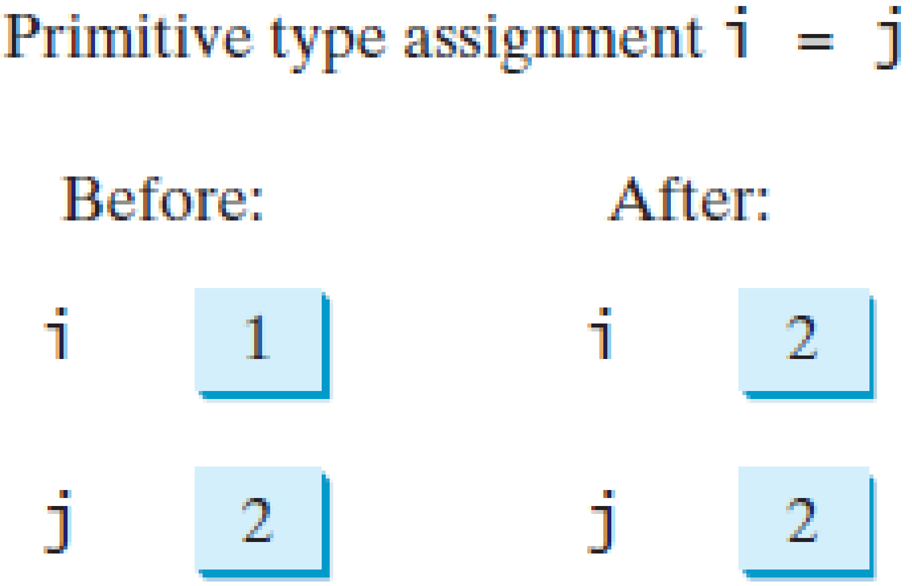
```
class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

Difference between Variables of Primitive Types and Reference Types

- Every variable represents a memory location that holds a value.
- 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 array or object is stored in memory.



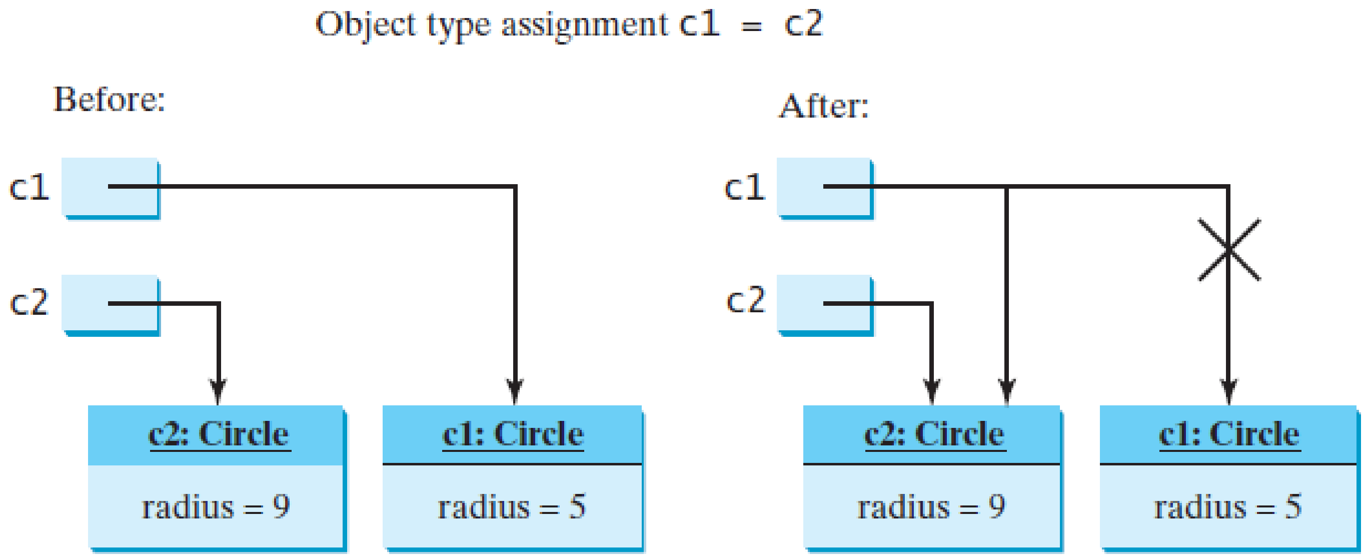
Difference between Variables of Primitive Types and Reference Types (Cont.)



Primitive type assignment `i = j`

Before:		After:	
i	1	i	2
j	2	j	2

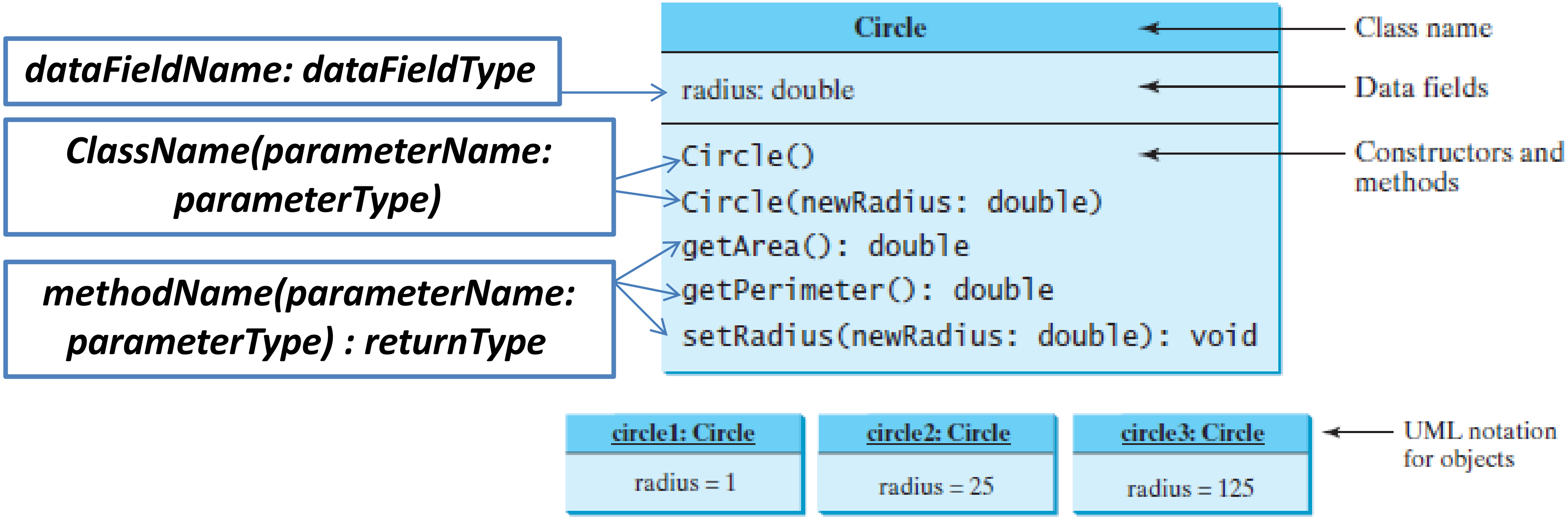
Difference between Variables of Primitive Types and Reference Types (Cont.)



- <u>After assignment:</u>
  - *c1* points to the same object referenced by *c2*.
  - The object previously referenced by *c1* is no longer useful and therefore is now known as *garbage*.
  - Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

UML Class Diagrams

- A standardized notation to illustrate classes and objects is the *Unified Modeling Language (UML)* class diagram.



dataFieldName: dataFieldType

ClassName(parameterName: parameterType)

methodName(parameterName: parameterType) : returnType