

Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Object-Oriented Programming II



CS2012: Introduction to Programming II

Reference Variables

Reference Variables

- ▶ objects in memory are accessed via ***reference variables***.

- does not contain the data for the object
- only contain a reference (memory address) to the object
- declared using the following syntax:

```
ClassName objectRefVar;
```

- ▶ all classes are ***reference types***

- a variable of the class type can reference an instance of the class.

```
Triangle myTriangle;
```

```
myTriangle = new Triangle(arguments);
```

- ▶ Can also be defined all on one line:

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

```
Triangle myTriangle = new Triangle();
```

Accessing an Object's Data and Methods

- ▶ Data can be accessed using the ***dot operator*** .
- ▶ `objectRefVar.dataField` references a (public) data field in the object.
 - `myTriangle.side1Length;`
- ▶ `objectRefVar.method(arguments)` invokes a method on the object
 - `myCircle.getArea()`
- ▶ `side1Length` is known as an ***instance variable*** since it is dependent on a specific instance of the class
- ▶ `getArea()` is an ***instance method*** because you invoke it only on a specific instance of the class
 - the object on which an instance method is invoked is the ***calling object***

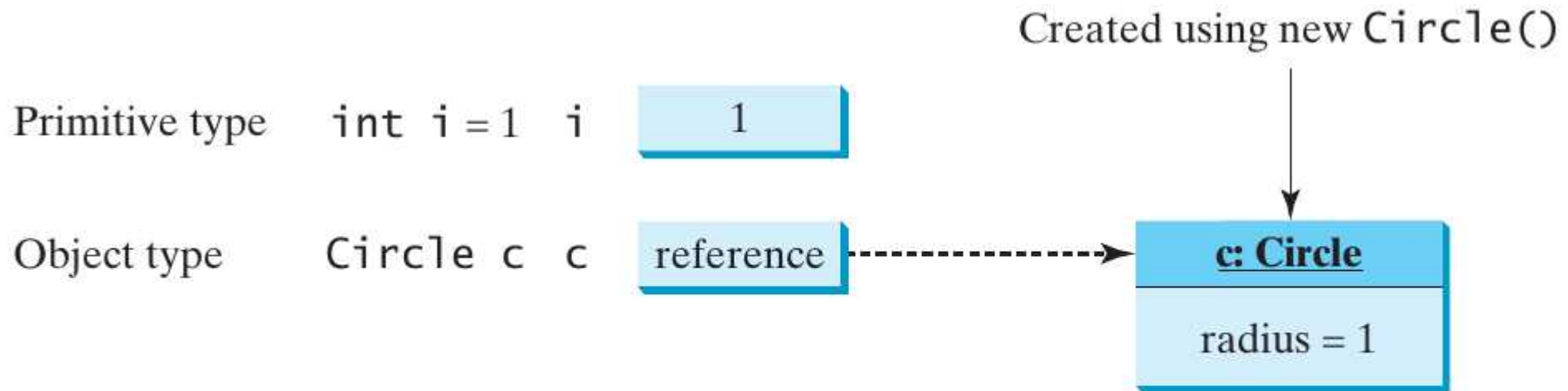
A Word of Caution

- ▶ Recall that you use:
 - `Math.methodName(arguments)`
 - `Math.pow(3, 2)`
 - to invoke a method in the `Math` class.
- ▶ Can you invoke `getArea()` using `Triangle.getArea()`?
 - The answer is no.
 - Some methods you saw before were ***static*** methods, which are defined using the `static` keyword.
 - `getArea()` is non-static and must be invoked through an object
 - ◆ `myTriangle.getArea();`

Primitive Types vs Reference Types

- ▶ Remember primitive types:
 - byte, short, int, long, float, double, boolean, char
- ▶ Every variable represents a memory location that holds a value
- ▶ When you declare a variable, you are telling the compiler the type of value the variable can hold
- ▶ For a variable of a primitive type, the value is of the primitive type
- ▶ For a variable of a reference type, the value is a reference (memory address) to where an object is located.

Primitive Types vs Reference Types



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.

Primitive Type Assignment Statements

- ▶ When you assign one primitive type variable to another primitive type variable, the value is **copied** from one to the other.
- ▶ Note: Changes to the value of *i* will NOT change the value of *j* and vice versa.

Primitive type assignment $i = j$

Before:

i 1

j 2

After:

i 2

j 2

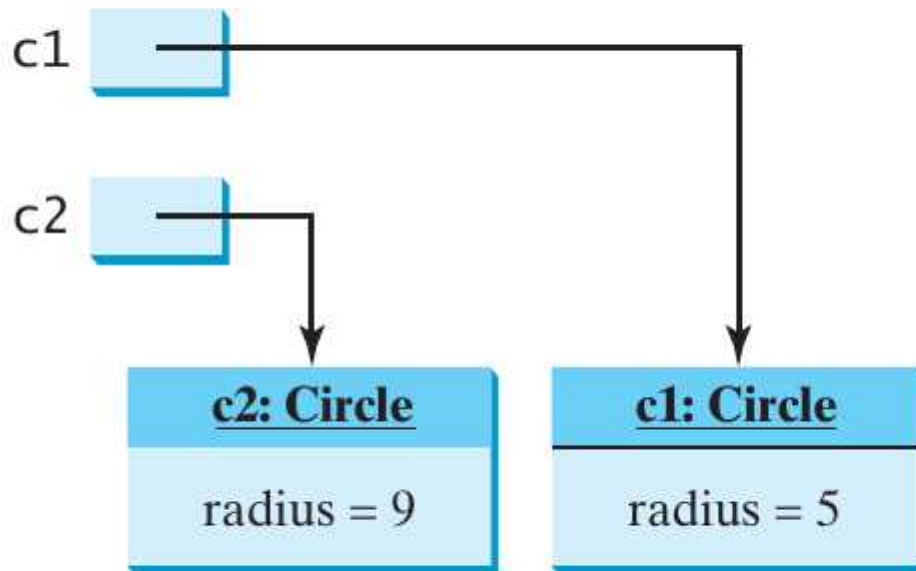
Primitive variable *j* is copied to variable *i*.

Reference Type Assignment Statements

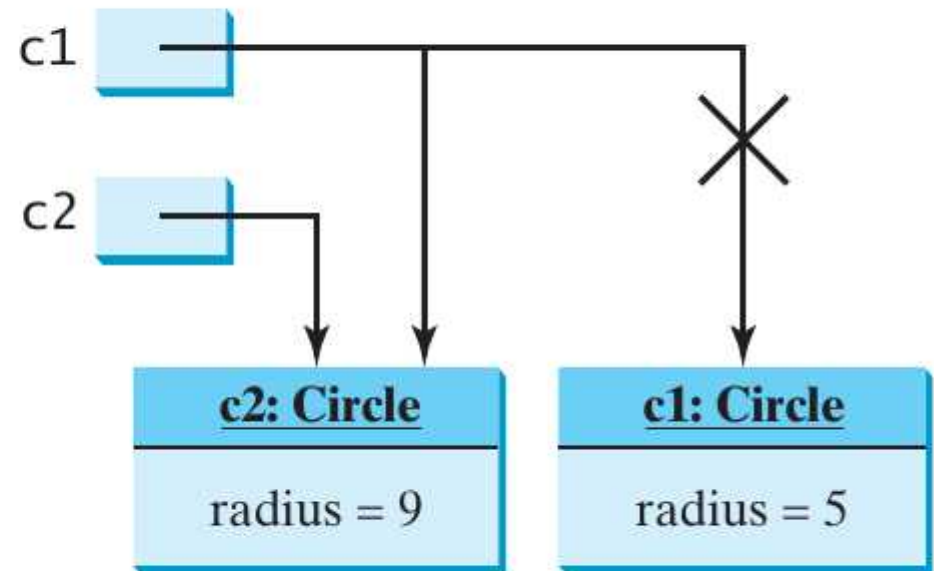
- ▶ For a reference variable, the reference of one variable is copied to the other variable.
- ▶ Note: Changes to the object c1 will also change the object c2

Object type assignment `c1 = c2`

Before:



After:



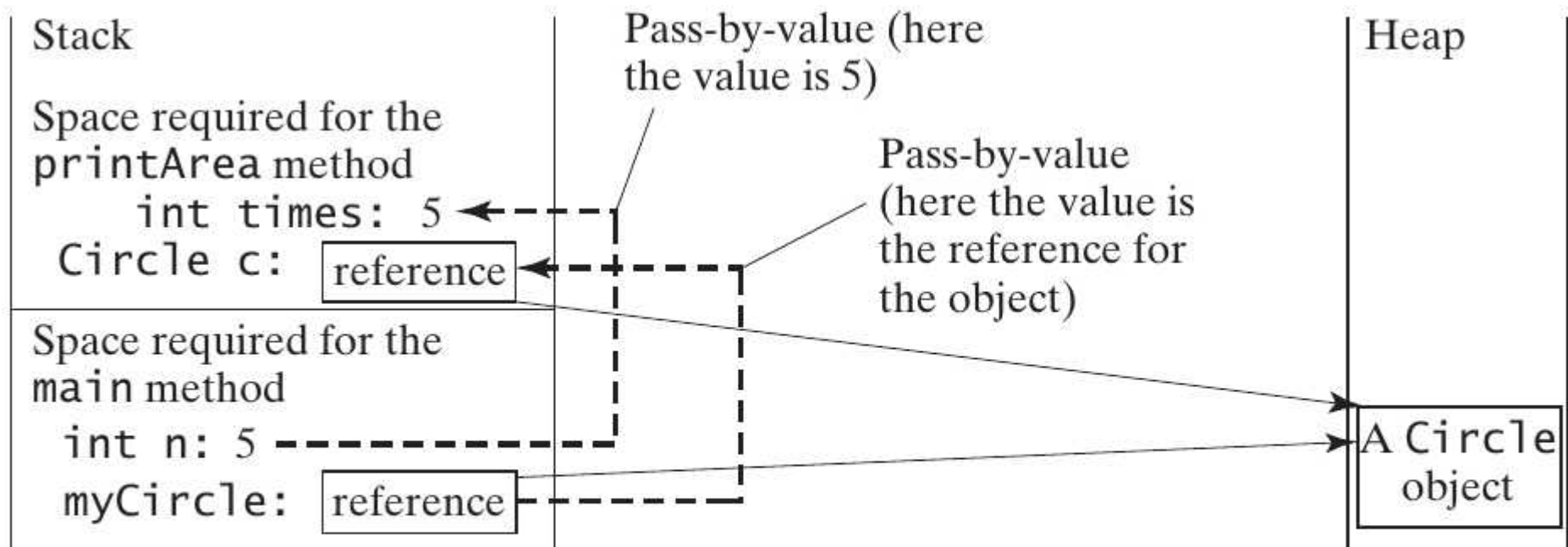
Reference variable `c2` is copied to variable `c1`.

Garbage Collection

- ▶ After the assignment statement `c1 = c2`:
 - `c1` points to the same object referenced by `c2`
 - The object previously referenced by `c1` now unreferenced.
- ▶ This object is now ***garbage***.
 - garbage is something that is taking up memory, but has no reference pointing to it.
 - the object is useless and cannot be used at all, we cannot access it in our program from this point forward.
- ▶ Garbage is automatically collected by JVM and the memory that the garbage object was occupying is freed up.
 - this is known as ***garbage collection***

Review: Stack and Heap

- ▶ Remember the ***stack*** is used to allocate memory for methods and local variables.
- ▶ Objects are created in an area of memory called the ***heap***.



The value of **n** is passed to **times**, and the reference to **myCircle** is passed to **c** in the **printAreas** method.

Static Variables, Constants, and Methods

Static Variables

- ▶ Instance variables
 - An instance variable is a data field which is tied to a specific instance of the class.
 - It is NOT shared among objects of the same class
- ▶ Example: the data field radius in the Circle class

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1.radius;
c2.radius;
```
- ▶ The value of radius in c1 is independent of the value of radius in c2

Static Variables

► Static variables

- Also called class variables
- Store values for the variables in a common memory location
- Shared by all objects of the same class
- All objects of the same class are affected if one object changes the value of a static variable

► Accessing static variables:

- Syntax: `ClassName.staticVariable`
- Example: `Math.PI`

Class Constants

- ▶ Constants in a class are shared by all objects,
 - constants should be declared `final static`
- ▶ Eg: the constant PI in the Math class is defined as:
 - `final static double PI = 3.14159265358979323846`

Static Methods

- ▶ Static methods
 - are not tied to a specific instance of object.
 - are NOT allowed to access instance data fields or instance methods
 - can ONLY access static data fields or static methods
- ▶ Declaration of static variables and methods
 - Use the modifier static
- ▶ Calling static methods
 - Syntax: `ClassName.methodName(arguments)`
 - Example: `Math.abs(-7)`
 - Non static variables CANNOT be used in Static methods.

Static Tip

- ▶ To improve the readability of your program
 - Use `ClassName.methodName(arguments)` to invoke a static method
 - Use `ClassName.staticVariable` to access a static variable
- ▶ Another programmer can easily recognize the static method and data in the class

Static vs. Instance Data Access

- ▶ The following tables summarizes what static and instance methods can access

	invoke instance methods	invoke static methods	access instance data fields	access static data fields
static methods		✓		✓
instance methods	✓	✓	✓	✓

Deciding on Static or Instance

- ▶ How do you decide to use a static or an instance method or variable?
 - a variable or method that depends on a specific instance of a class should be an instance variable or method
 - a variable or method that does not depend on a specific instance of a class should be a static variable or method
 - constants should always be declared as **final** and **static**.
- ▶ Example:
 - every circle has its own radius, so radius should be an instance variable.
 - every circle has its own area so the `getArea()` method should be an instance method.
 - None of the methods in the `Math` class depend on a specific instance of that class, so all of these methods are static
 - The `main` method never depends on a specific instance of a class and it is also static.

Access (Visibility) Modifiers

Access (Visibility) Modifiers

- ▶ ***access modifiers (visibility modifiers)***: specify how data fields and methods in a class can be accessed from outside of the class
 - **public**: The class, data fields, or method is accessible from any class in any package
 - **private**: The class, data fields, or methods are accessible only from within their own class
 - **protected**: The class, data fields, or methods are accessible by anything in the same package.
 - **no visibility modifier**: this is also called ***package-private*** or ***package-access***, and classes methods and data fields are accessible by any class in the same package, even the default package.

Arrays as Data Fields

Arrays as Data Fields

- ▶ Remember that any data type / object can be used as a data field inside a class.
- ▶ This also includes arrays.
- ▶ Arrays can be used to store a list of information about your object.
- ▶ Getters and Setter for array data fields:
 - Be careful with how you write your getters and setters.
 - Remember that arrays are references, so any changes to the array outside of the object, will keep the changes inside the object as well.
- ▶ For data safety reasons, you may want to make your array "immutable"

Arrays and ArrayLists of Objects

Array of Objects

- ▶ Multiple instances of a class can be stored inside of an array.
- ▶ Example: The following array will accept Triangle objects.
 - `Triangle[] triangleArray = new Triangle[10]`
- ▶ Each position in the array is a reference pointing to an object in memory.
 - You can use the dot operator on a specific position of the array to access member items of your object.
 - Example:
 - ◆ `triangleArray[5].area()`

Array of Objects

```
public static void main(String[] args) {  
    //Create our array  
    Triangle[] triangles = new Triangle[3];  
  
    //Create some Triangle objects  
    Triangle t1 = new Triangle(5, 10, 15);  
    Triangle t2 = new Triangle(4, 3, 4);  
    Triangle t3 = new Triangle(5, 5, 5);  
  
    //Store the triangles in the array;  
    triangles[0] = t1;  
    triangles[1] = t2;  
    triangles[2] = t3;  
  
    //Print out the area of each Triangle by looping  
    //through the array  
    for(int i = 0 ; i < triangles.length ; i++) {  
        System.out.println( triangles[i].area() );  
    }  
}
```

Array of Objects

```
public static void main(String[] args) {  
    //Create our array  
    Triangle[] triangles = new Triangle[3];  
  
    //Create some Triangle objects, store them in an array  
    //all in one step.  
    int side1 = 4;  
    int side2 = 2;  
    int side3 = 7;  
    for (int i = 0 ; i < triangles.length ; i++) {  
        triangles[i] = new Triangle(side1, side2, side3);  
        side1 += 2  
        side2 += 1  
        side3 += 4  
    }  
  
    //Print out the area of each Triangle by looping  
    //through the array  
    for(int i = 0 ; i < triangles.length ; i++) {  
        System.out.println( triangles[i].area() );  
    }  
}
```

ArrayList of Objects

- ▶ ArrayLists are similar to arrays but have some added features:
 - ArrayLists are part of the Java Collections Classes
 - ArrayLists are implemented using an array "under the hood"
 - ArrayLists can change their sizes dynamically during runtime (remember that arrays cannot do this)
 - ArrayLists have many built-in methods for easy access.
 - You will learn ArrayLists in more detail in CS-2013, but for now they will be very useful for this class.
 - ArrayLists can only hold Object types, it cannot hold primitives.
- ▶ ArrayLists have to be imported
 - `import java.util.ArrayList;`

ArrayList of Objects

▶ Creating an ArrayList Syntax:

```
ArrayList<Type> listName = new ArrayList<Type>();
```

- ArrayList is created just like Classes and Objects you have already seen with the new keyword.
- <Type> is required and you have to tell the compiler what Type of object the ArrayList will be storing
 - ◆ this is related to **generics** which you will learn more about in CS-2013

▶ Example:

```
ArrayList<Triangle> tris = new ArrayList<Triangle>();
```

ArrayList API

- ▶ You should read through the ArrayList API online to learn more about how to use the Class.
- ▶ ArrayList API

Immutable Objects and Classes

Immutable Objects and Classes

- ▶ Sometimes you may want an object whose data cannot be changed once it has been created.
 - this is called an *immutable object* and its class an *immutable class*
- ▶ The `String` class is such a class
 - once a string is created, its content cannot be changed.
 - changing a string actually involves creating a new string object in memory
- ▶ For a class to be immutable, it must follow these rules:
 - all data fields must be private
 - there can't be any mutator methods for data fields
 - no accessor methods can return a reference to a data field that is mutable.

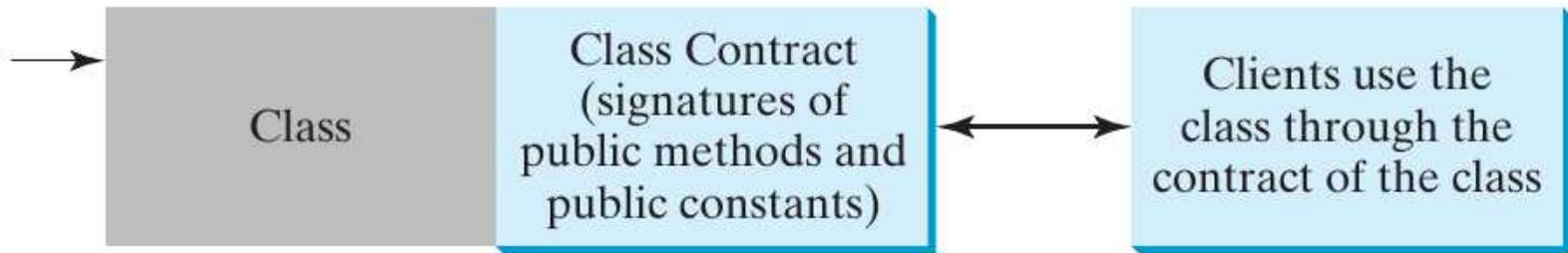
Abstraction

Abstraction

- ▶ ***abstraction*** means separating the class implementation (how the class actually works) from how the class is used.
 - When you create an instance of Scanner, or when you use the Math class, you don't know exactly how Math or Scanner work (are implemented) but you still know how to use them.
- ▶ ***public interface of a class (class contract)***: the collection of methods and data fields that are accessible from outside of the class (public methods and / or data fields) coupled together with the description of how these members behave.
- ▶ A user of a class doesn't (and shouldn't) need to know HOW the class actually works "under the hood".
- ▶ We say the details of implementation are ***abstracted*** and hidden from the user of the class.

Abstraction

Class implementation
is like a black box
hidden from the clients



Class abstraction separates class implementation from the use of the class.

UML Diagrams

UML Diagrams

- ▶ Classes can be designed using a UML (Unified Modeling Language) Diagram.
 - data fields are denoted as:
 - ◆ `dataFieldName: dataFieldType`
 - constructors are denoted as:
 - ◆ `ClassName(parameterName: parameterType)`
 - methods are denoted as:
 - ◆ `methodName(parameterName: parameterType): returnType`

UML Diagrams

Circle

-radius: double

-numberOfObjects: int

+Circle()

+Circle(radius: double)

+getRadius(): double

+setRadius(radius: double): void

+getNumberOfObjects(): int

+getArea(): double

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.

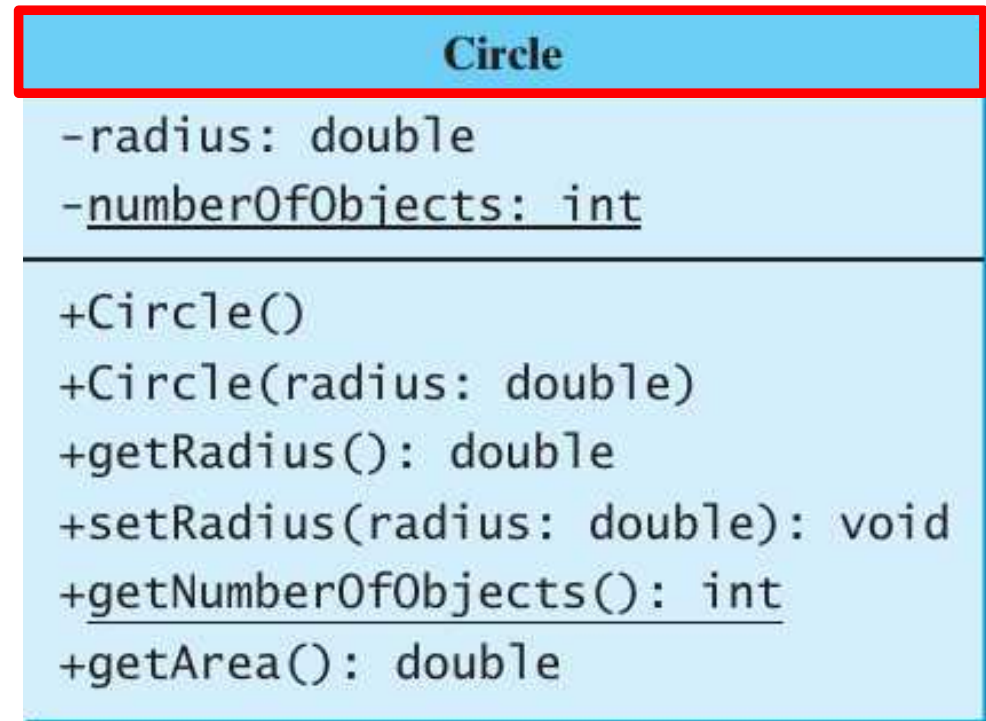
Sets a new radius for this circle.

Returns the number of circle objects created.

Returns the area of this circle.

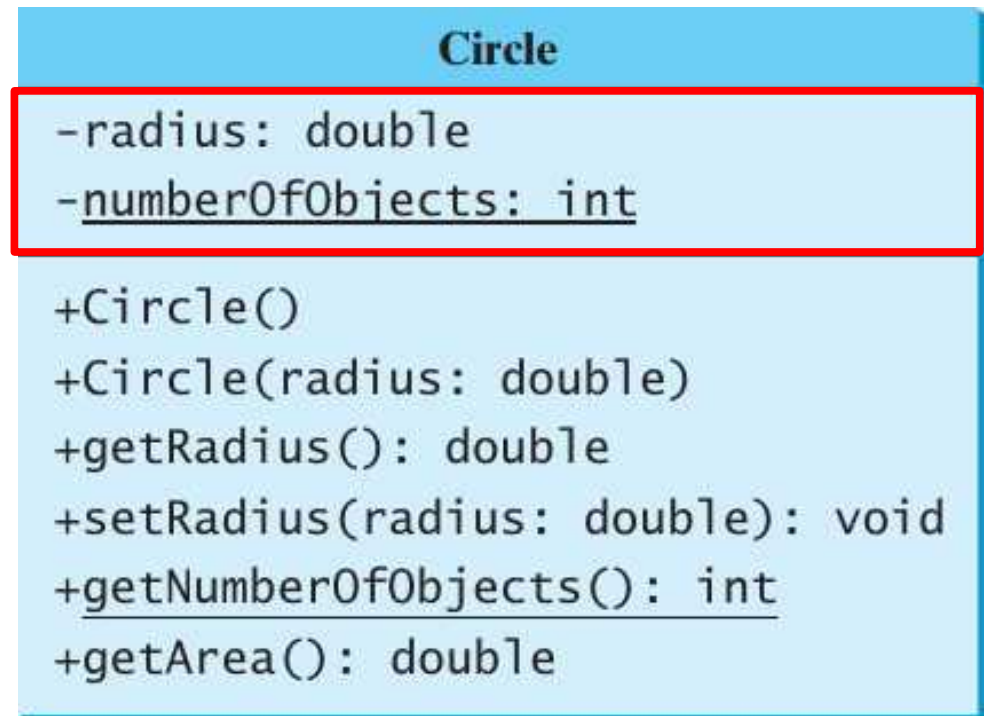
UML Diagrams

- ▶ Class name goes at the top of the UML diagram.
- ▶ Class name should be centered.
- ▶ Draw a line separating the class name from the data fields.



UML Diagrams

- ▶ Data fields should be listed next.
 - dataFieldName: datatype
- ▶ Indicate access modifier
 - + for public
 - - for private
 - # for protected
- ▶ Underline any static data fields
- ▶ Draw a line between the data fields and the methods



UML Diagrams

- ▶ Constructors should be listed first.

ClassName(name1: type1,
name2: type2, etc.)

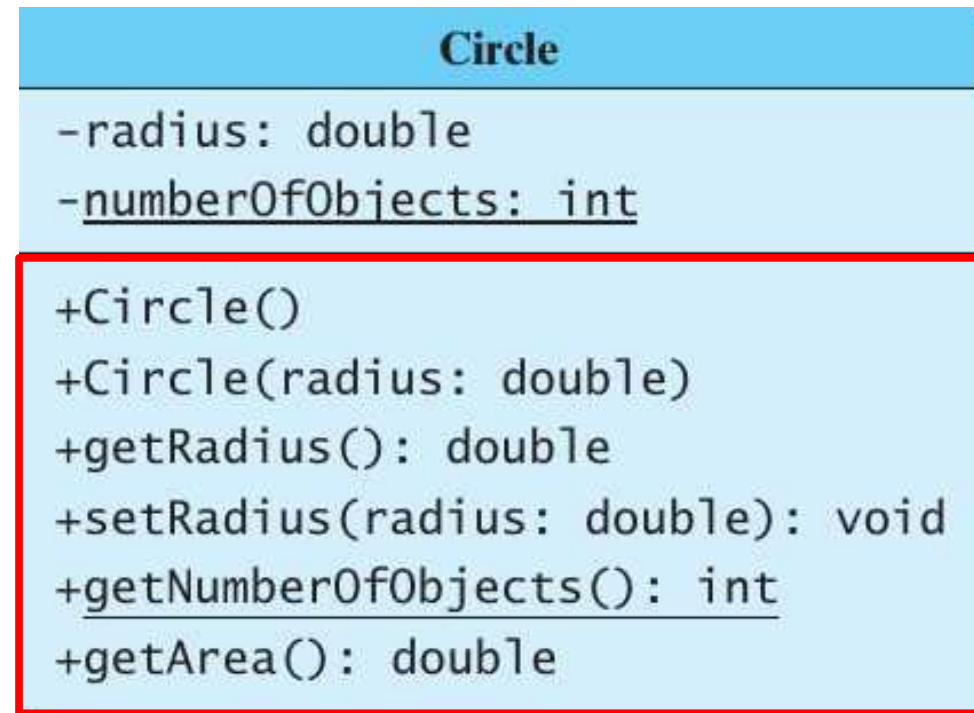
- list all parameters and their data types
- indicate access modifier

- ▶ Methods come next

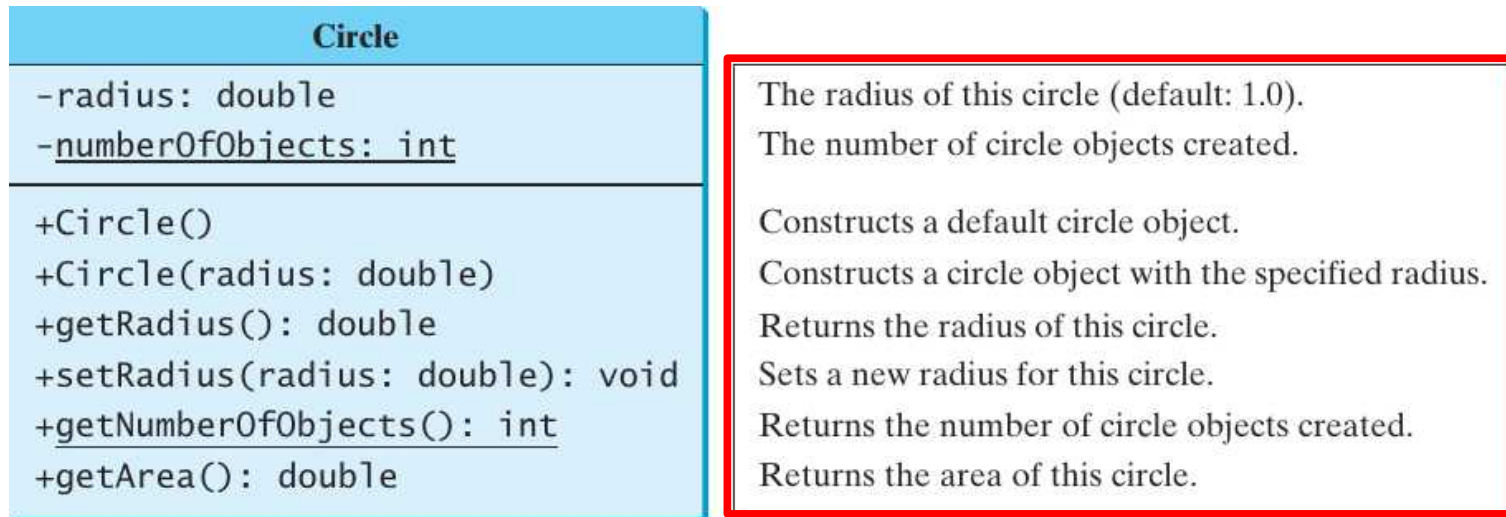
methodName(name1: type1, name2:
type2, etc.): returnType

- list all parameters
- indicate the return type
- indicate access modifier

- ▶ Underline any static methods





UML Diagrams



- ▶ Data Fields:
 - Give a one sentence description of the data field.
 - Indicate if the data field has a default value or not
- ▶ Constructors:
 - Give a one sentence description of what that constructor does.
- ▶ Methods:
 - Give a one sentence description of what that method does.

Wrapper Classes

- 
- ▶ Why doesn't Java process primitive type values as Objects?
 - objects cause overhead in programming
 - primitive type operations are generally faster
 - Java's performance would be adversely affected if primitive data type values were treated as objects.
 - ▶ Caveat: Many Java methods and data structures require the use of objects as arguments (ArrayLists can only store objects, not primitive types.)
- 

Wrapper Classes

- ▶ Java provides a wrapper class for each of the primitive types:
 - Boolean, Character, Double, Float, Byte, Short , Integer, Long
 - these are in the java.lang package
 - each of these are used to "wrap" primitive type values and turn them into Object types

Wrapper Objects

- ▶ Wrapper classes do not have no-arg constructors
- ▶ All instances of wrapper classes are immutable
 - internal values cannot be changed once the objects are created.

MIN and MAX Constants

- ▶ All numeric wrapper classes have constants to get the maximum and minimum possible values that data type can hold.
 - `MAX_VALUE` for the max
 - `MIN_VALUE` for the min
- ▶ Examples: All min and max constants should be prefixed with the class name:
 - `Integer.MAX_VALUE` and `Integer.MIN_VALUE`
 - `Double.MAX_VALUE` and `Double.MIN_VALUE`
 - and so on...

Conversion Methods

- ▶ Each numeric wrapper class has methods which convert from one numeric type to another:
 - `doubleValue()` returns a double
 - `floatValue()` returns a float
 - `intValue()` returns an int
 - `longValue()` returns a long
 - `shortValue()` returns a short

Comparing Wrapper Classes

- ▶ Numeric Wrapper classes also contain a `compareTo()` method which compares two numbers.
 - returns 1 if the first number is greater than the second
 - returns 0 if the first number is equal to the second
 - returns -1 if the first number is less than the second

Converting Strings to Numbers

- ▶ Numeric wrapper classes have a static method `valueOf(String s)` which creates a new object initialized to the value represented by the string.

Conversion Between Primitives and Wrappers

- ▶ Primitive values can be automatically converted to an object using a wrapper class, and vice versa
- ▶ **boxing**: converting a primitive value to a wrapper object
- ▶ **unboxing**: converting a wrapper object to a primitive value.
- ▶ compiler automatically boxes a primitive value that appears in a context requiring an object (**autoboxing**) and will unbox an object that appears in a context requiring a primitive value (**autounboxing**)

BigInteger and BigDecimal

- ▶ `BigInteger` and `BigDecimal` can be used to represent integers or decimal numbers of any size or precision.
- ▶ You would use these when the `long` or `double` types are not enough.
- ▶ For more information read about them in the Java API.

References

- ▶ Liang, Chapter 09: Classes and Objects
- ▶ Liang, Chapter 10: Thinking in Objects