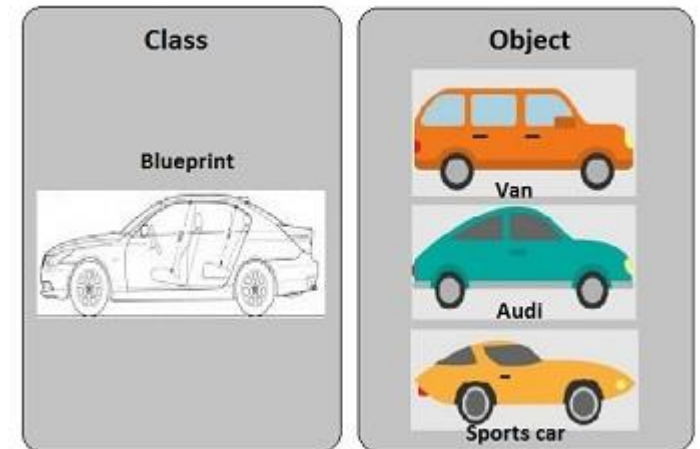


Objects and Classes



Shing-Chi Cheung
Computer Science and Engineering
HKUST



Classes

- A class **defines** objects of the same **type**
 - Java is a strongly typed language
 - Benefit: many coding mistakes can be exposed as type errors in compilation
- A class uses **variables** to define data fields and **methods** to define behaviors
- A class provides **constructors** to create objects from the class



Classes

A class is a collection of:

- Attributes
 - Instance **variables**
 - Static **variables**
- Operations
 - Instance **methods**
 - Static **methods**

```
public class Circle {  
    private double radius; // instance variable  
  
    public Circle(double radius) { // constructor  
        this.radius = radius;  
    }  
  
    public Circle() { // no-arg constructor  
        this(1.0); // call the constructor above  
    }  
  
    public double getArea() { // instance method  
        return this.radius * this.radius * Math.PI;  
    }  
}
```



Classes

A class is a collection of:

- Attributes
 - **Instance variables**
 - Static variables
- Operations
 - **Instance methods**
 - Static methods

```
public class Circle {  
    private double radius; // instance variable  
  
    public Circle(double radius) { // constructor  
        this.radius = radius;  
    }  
  
    public Circle() { // no-arg constructor  
        this(1.0); // call the constructor above  
    }  
  
    public double getArea() { // instance method  
        return this.radius * this.radius * Math.PI;  
    }  
}
```



Accessing Instance Methods and Variables

- Each instance methods or instance variables are exclusively **owned by an object** (a.k.a. instance)
- Access them via their owner object
 - ❑ `Circle c = new Circle(); c.getArea();` // before Java 10
 - ❑ `var c = new Circle(); c.getArea();` // via a circle referenced by c
 - ❑ `new Circle().getArea();` // via an anonymous circle object
 - ❑ `this.getArea();` // via the current object

var is a handy way to declare a **local variable** whose type can be inferred automatically by compiler from the variable's assigned value.



var helps decouple client code from library



Library code

```
package tlib;

public class Table {
    ..
    int lookup(int key) {
        ...
    }
}
```

changed to another type, say Double

*We will see more examples of using **var**.*

Client code

```
import tlib.Table;

...
void search(Table t, int key) {
    var v = t.lookup(key);
    ...
}
```

no change is needed at client

this Keyword



■ **this** references the **current** object.

□ A common use of **this** is to reference the **current object's** instance variables.



■ **this** may be used by a constructor to invoke **another constructor** of the same class.

```
public class Circle {  
    private double radius; // instance variable  
  
    public Circle(double radius) { // constructor  
        this.radius = radius;  
    }
```

```
    public Circle() { // no-arg constructor  
        this(1.0); // call the constructor above  
    }
```

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

Usually omit **this** when the current method and the referenced var belong to the same instance

Classes

A class is a collection of:

- Attributes

- Instance variables
- **Static variables**

- Operations

- Instance methods
- **Static methods**

```
public class Circle {  
    private double radius; // instance variable  
  
    public Circle(double radius) { // constructor  
        this.radius = radius;  
    }  
  
    public Circle() { // no-arg constructor  
        this(1.0); // call the constructor above  
    }  
  
    public double getArea() { // instance method  
        return this.radius * this.radius * Math.PI;  
    }  
}
```



Note: Static variables and methods are also known as class variables and methods

Accessing Static Methods and Variables

- Static methods and static variables are **owned by a class**
- They are **shared** by all instances created from their owner class
- Access them via their owner class or an instance of the class
 - `Circle.numOfObjects;` *// via class*
 - `new Circle().numOfObjects;` *// via instance*
 - `Circle.main(null);` *// via class*

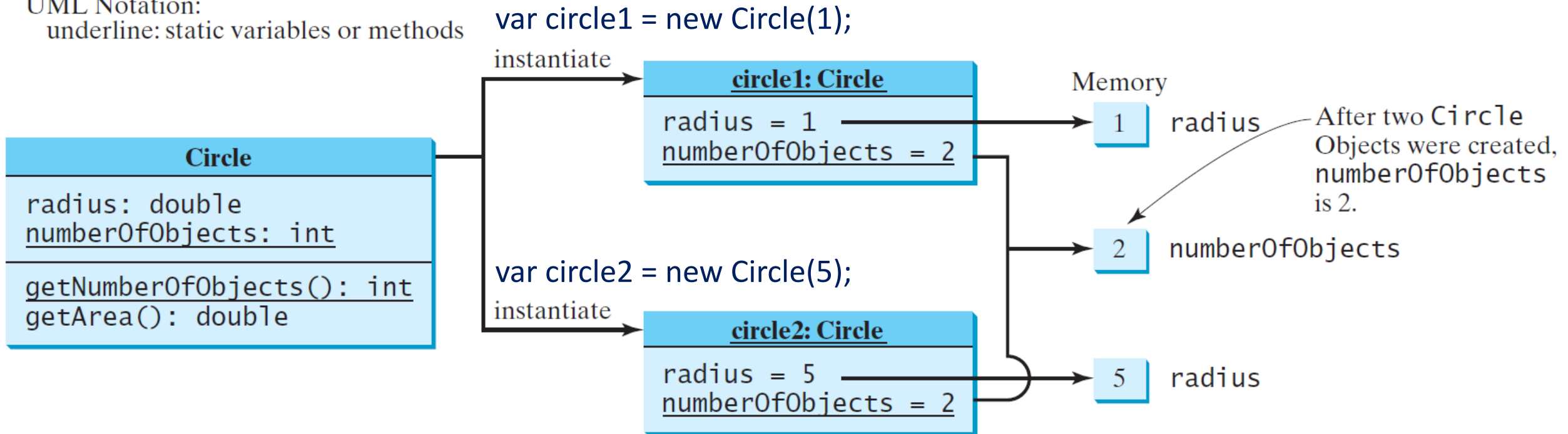
```
public class Circle {  
    public static int numOfObjects;  
    ...  
    public static void main(String[] args) {  
        ...  
    }  
}
```

[CircleWithStaticMembers.java](#)
[TestCircleWithStaticMembers.java](#)
[StaticInstanceExample.java](#)

Static Variables, Constants and Methods

UML Notation:

underline: static variables or methods



Comparison: Instance versus Static

Instance Variable and Method

- An **instance variable** belongs to a specific instance
- An **instance method** is invoked by an instance of its class

Static Variable, Method and Constant

- A **static variable** is shared by all the instances of its class
- A **static method** is not tied to a specific instance.
- A **constant** is a **static final variable** shared by all the instances of its class

Reference Variables and Default Initialization

- Instance and static variables are initialized with a **default** value 

```
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'  
}
```

[DefaultValueTest.java](#)

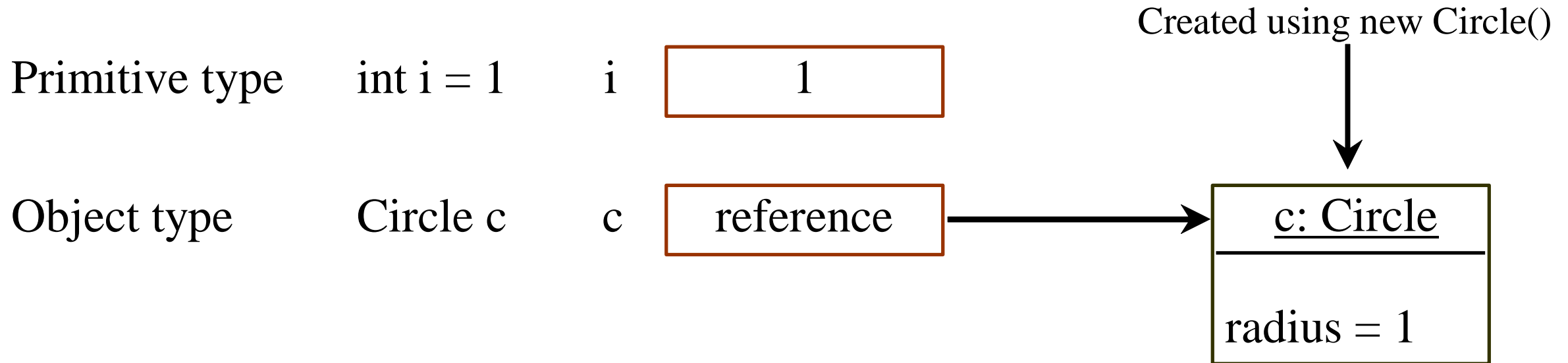
NO default value is assigned to a local variable

```
public class NoDefault4Locals {  
    public static void main(String[] args) {  
        int x; // x does not have a default value  
        String y; // y does not have a default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```

Compile error: variable not initialized

[NoDefault4Locals.java](#)

Differences between Variables of Primitive Data Types and Reference Data Types



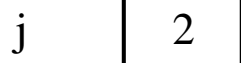
Copying Variables of Primitive Data Types and Object Types

`i = j; // primitive type assignment`

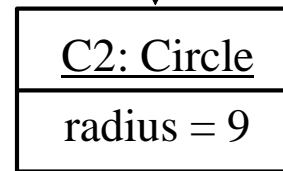
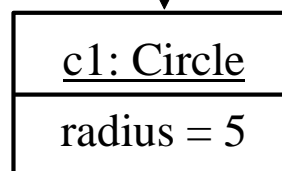
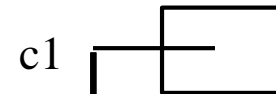
Before:



After:



Before:

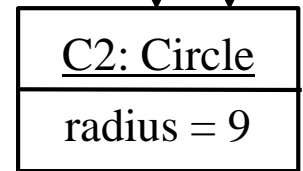
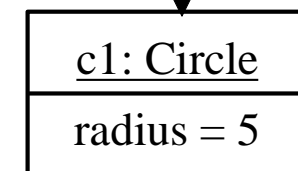
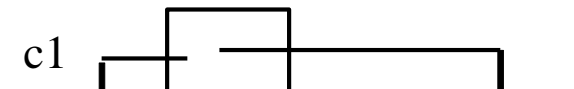


`Circle c1 = new Circle(5);`


`Circle c2 = new Circle(9);`

`c1 = c2; // obj type assignment`

After:



Default Constructor

- A *default constructor* is provided automatically *only if no constructors are explicitly defined in a class* 
- Default constructor takes **no arguments** and has an **empty body**

```
class DefConstr {  
}
```

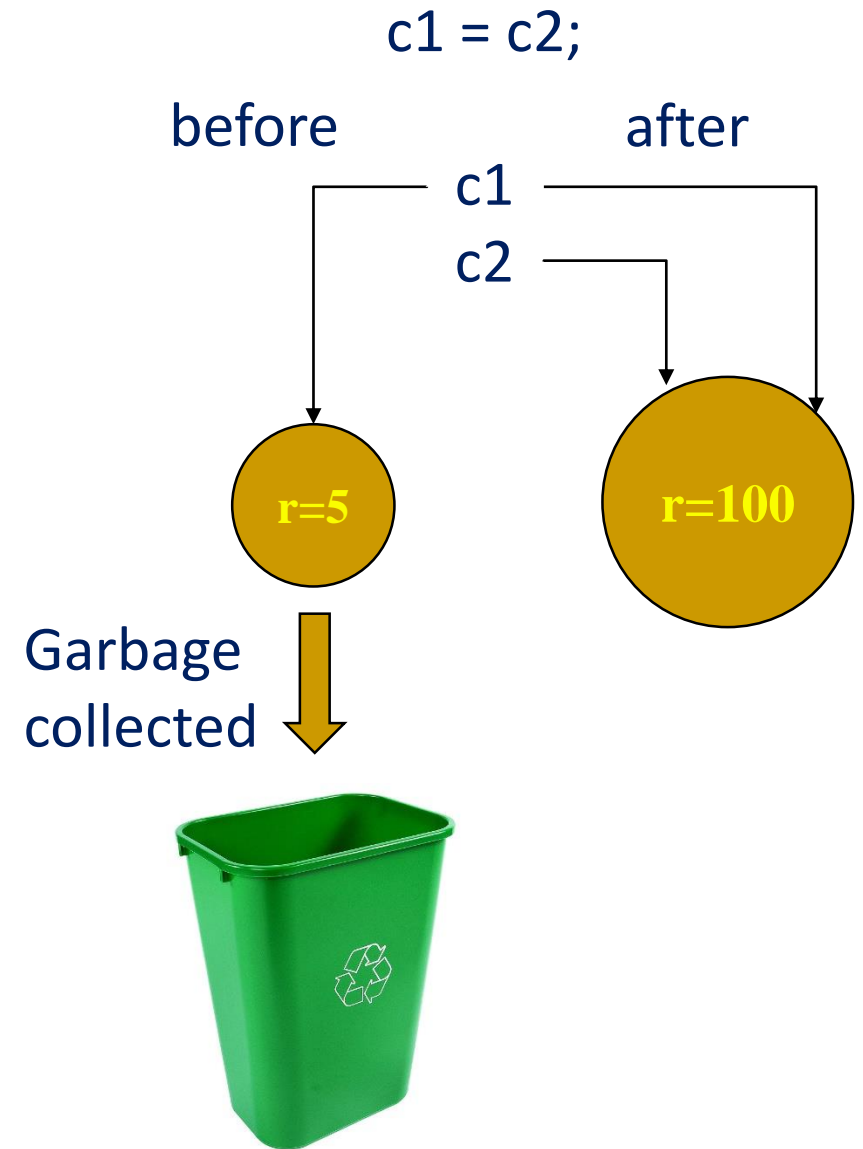


```
class DefConstr {  
    // default constructor  
    DefConstr() { }  
}
```

[DefConstr.java](#)

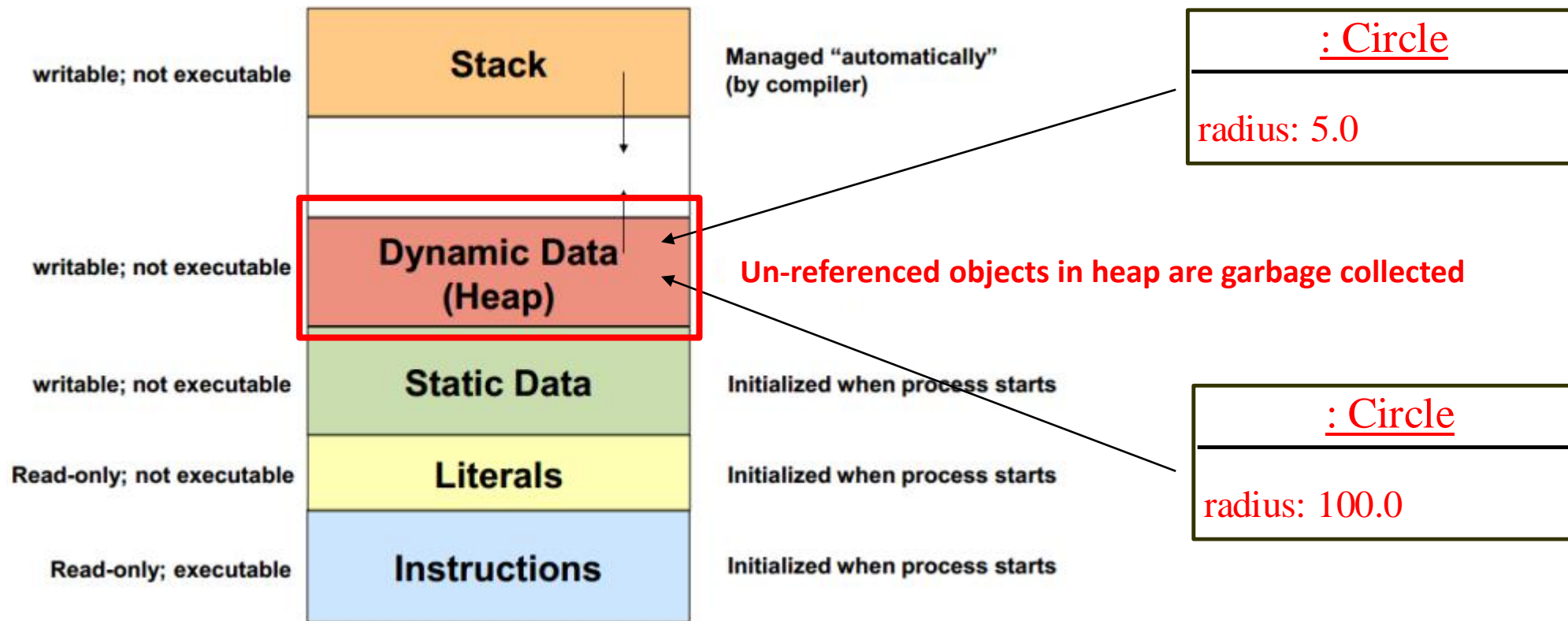
Garbage Collection

- After the assignment, the circle object (with radius = 5) is **no longer referenced** by any variables. It is considered a **garbage**
- Garbage's memory will be **automatically recycled** by JVM




Garbage Collection

Java objects are **automatically** managed by JVM in the **heap**.
We need not explicitly allocate/deallocate memories for objects



Garbage Collection, cont

- JVM **automatically** releases the memory of objects that are not referenced by variables
- TIP: If you no longer use an object, you should assign **null** to its reference variable 
- Note: Java 14 supports multiple implementations of garbage collectors to suit different needs
 - ❑ <https://docs.oracle.com/en/java/javase/14/gctuning/available-collectors.html#GUID-F215A508-9E58-40B4-90A5-74E29BF3BD3C>
 - ❑ ZGC (<https://wiki.openjdk.java.net/display/zgc/Main>)

Using Java Library 1 - Date

- `java.util.Date`: a built-in class for date and time
- Use the `Date` class to create an instance of date and time
 - `Date date = new Date();`
- Use the created instance's `toString` method to print the date and time as a string
 - `System.out.println(date);`
 - `"Tue Aug 14 21:04:22 CST 2018"`
- API documentation
 - <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Date.html>
- Superseded by `java.time.{LocalDateTime, LocalDate, LocalTime, Instant}`

<code>java.util.Date</code>
<code>+Date()</code> <i>// since 1/1/1970</i>
<code>+Date(elapseTime: long)</code>
<code>+toString(): String</code>
<code>+getTime(): long</code>

[DateExampleWithoutImport.java](#)
[DateExampleWithImport.java](#)

Using Java Library Class 2 - Random

Use `Math.random()` to obtain a random double value between 0.0 and 1.0 (excluding 1.0). A more powerful random number generator is provided in the `java.util.Random` class

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

The Random Class Example

If two **Random** objects have the same seed, they generate identical sequences of numbers. E.g., the following code creates two **Random** objects with the same seed 3

Console output:

From random1: 734, 660, 210,
581, 128, 202, 549, 564, 459,
961

From random2: 734, 660, 210,
581, 128, 202, 549, 564, 459,
961

```
Random random1 = new Random(3); // seed = 3
System.out.print("From random1: ");
for (var i = 0; i < 9; i++)
    System.out.print(random1.nextInt(1000) + ", ");
System.out.print(random1.nextInt(1000));
```

```
Random random2 = new Random(3); // seed = 3
System.out.print("\nFrom random2: ");
for (var i = 0; i < 9; i++)
    System.out.print(random2.nextInt(1000) + ", ");
System.out.print(random2.nextInt(1000));
```

Using Java Library Class 3 - Point2D

- Java API has a convenient **Point2D** class in the **javafx.geometry** package for representing a point in a two-dimensional plane

javafx.geometry.Point2D

```
+Point2D(x: double, y: double)
+distance(x: double, y: double): double
+distance(p: Point2D): double
+getX(): double
+getY(): double
+toString(): String
```

Constructs a `Point2D` object with the specified x - and y -coordinates.
Returns the distance between this point and the specified point (x, y) .
Returns the distance between this point and the specified point p .
Returns the x -coordinate from this point.
Returns the y -coordinate from this point.
Returns a string representation for the point.

[TestPoint2D.java](#)

The Point2D Class (console output)

Output:

Enter point1's x-, y-coordinates: 1 4

Enter point2's x-, y-coordinates: 2 5

p1 is Point2D [x = 1.0, y = 4.0]

p2 is Point2D [x = 2.0, y = 5.0]

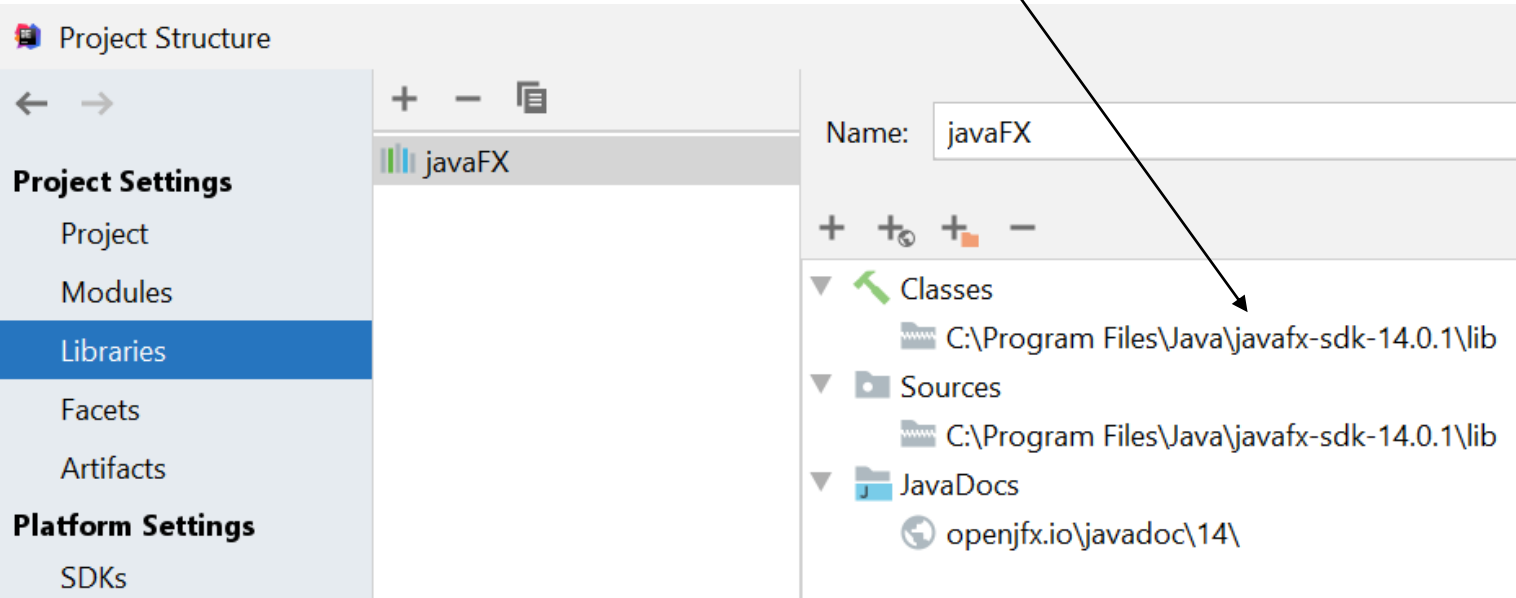
The distance between p1 and p2 is 1.4142135623730951

The midpoint between p1 and p2 is Point2D [x = 1.5, y = 4.5]

Using JavaFX in IntelliJ

Create a JavaFX library for your project:

Go to File -> Project Structure -> Libraries and add the JavaFX 14 SDK as a library to the project. Specify the path to the lib folder of the JavaFX SDK.



Starting from Java 11, JavaFX is no longer included in SDK

Install a popular JavaFX 14 SDK available separately at

<https://gluonhq.com/products/javafx/>

Optionally install a popular SceneBuilder available at

<https://gluonhq.com/products/scene-builder/>

<https://openjfx.io/openjfx-docs/#install-java> (pay attention to the section on JavaFX and IntelliJ)

Visibility Modifiers and Accessor/Mutator Methods

❑ public

- ❑ The class, variable or method is visible to **any class in any package**.

❑ protected

- ❑ Will be discussed later.

❑ No access modifier (called default)

- ❑ The class, variable or method is visible to **any class under the same package**.

❑ private

- ❑ The class, variable or method can be accessed only by the **declaring 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();
    }
}
```

```
package p1;

class C1 {
    ...
}
```

```
package p1;

public class C2 {
    can access C1
}
```

```
package p2;

public class C3 {
    cannot access C1;
    can access C2;
}
```

The private modifier restricts **access to within a class**, the default modifier restricts **access to within a package**, and the public modifier enables **unrestricted access**.



Public members in non-public classes?

```
package system;  
abstract class Prototype {  
    public static final int HASHKEY = 99881;  
}
```

Compilation errors?

Make sense



```
package user;  
import system.Prototype;  
public class Consumer {  
    public static void main(String[] args) {  
        System.out.printf("%d\n", Prototype.HASHKEY);  
    }  
}
```

Public members in non-public classes?

```
package system;  
abstract class Prototype {  
    public static final int HASHKEY = 99881;  
}  
public class Preview extends Prototype {  
    ... // preview functionalities  
}
```

Preview serves as an agent
to access HASHKEY

Make sense



```
package user;  
import system.Preview;  
public class Consumer {  
    public static void main(String[] args) {  
        System.out.printf("%d\n", Preview.HASHKEY);  
    }  
}
```

Passing Values to Method Parameters

- ❑ **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)
- ❑ Note: In Java terminologies, there is no passing by reference

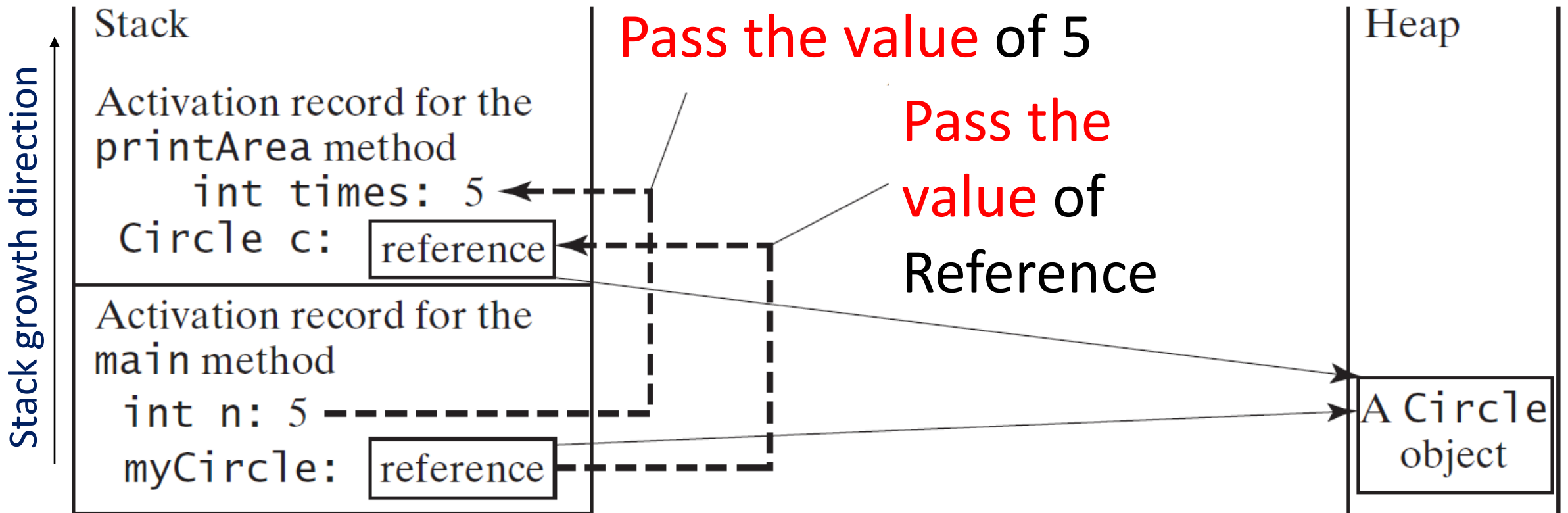
Console output:

Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483

Radius is 6.0
n is 5

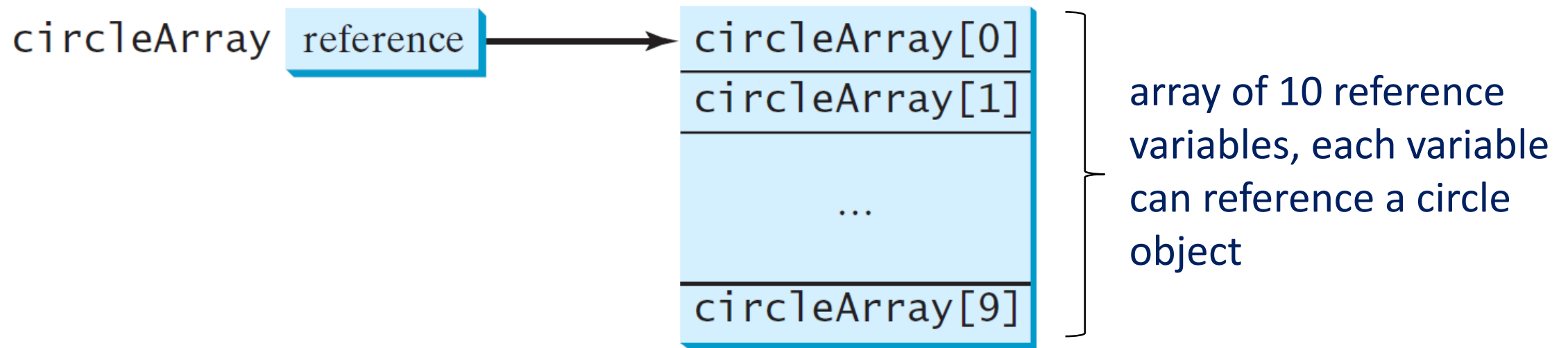
TestPassObjects.java

Passing Values to Method Parameters



Array of Objects

- `Circle[] circleArr = new Circle[10];`
- `Circle[]` denotes the type of Circle array
- An array of objects is actually an array of reference variables



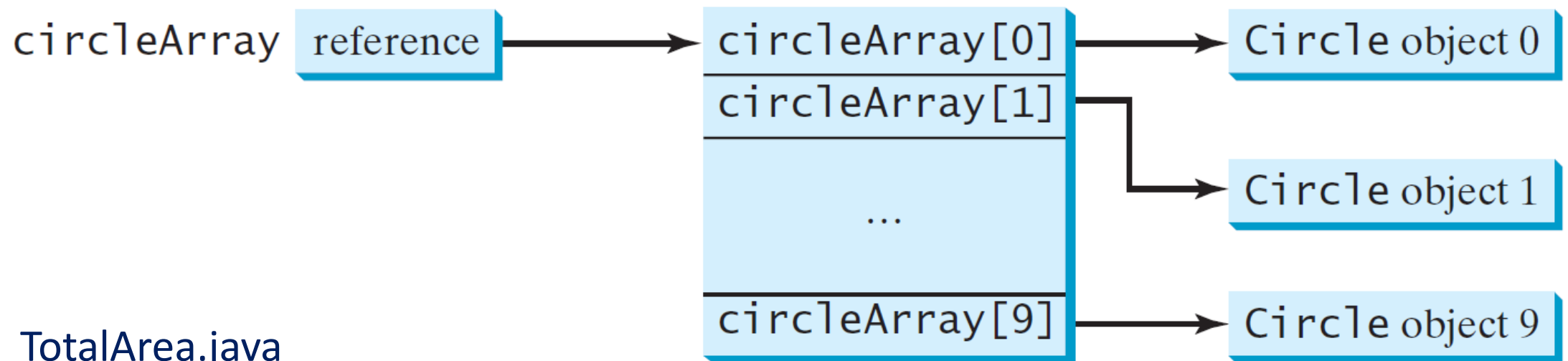
Local-Variable Type Inference (since Java 10)

- **var** circleArr = **new** Circle[10]; *// type inferred automatically*
 - instead of: Circle[] circleArr = **new** Circle[10];
- Java supports automatic type inference of **local variables**
- Useful in lambda expressions (discussed later)
- To display the inferred type, press CTL-Q (Win/Linux) or CTL-J (Mac) in IntelliJ

TotalArea.java

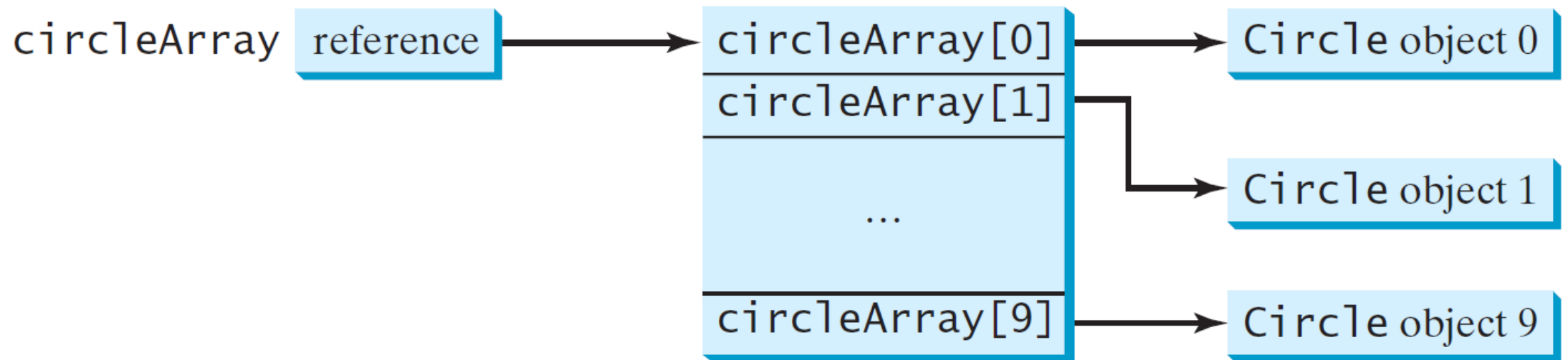
Array of Objects, cont.

- Invoking `circleArr[1].getArea()` involves two levels of referencing
 - `circleArray` references the entire array.
 - `circleArray[0]` references the first indexed `Circle` object.



Array of Objects

```
var circleArr = new Circle[10];  
for (var i = 0; i < circleArr.length; i++)  
    circleArr[i] = new Circle(Math.random()*100);
```



Array of Objects, cont.

Default initialization of array elements

Element Type	Default Initialization
int	0
byte	0
double	0.0
boolean	false
Object reference	null

The ArrayList Class

- An **array** object's size is **fixed** once it is created
- An **ArrayList** object's size can be **adjusted** after it is created

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

<E> declares a **type parameter E**. Assign E to the type of objects stored by the ArrayList; otherwise any types of objects can be stored.

Creates an empty list.

Appends a new element o at the end of this list.

Adds a new element o at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element o.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element o from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.

Generic Type

Note: We will discuss generic class and type parameter in a separate topic later.

- **ArrayList** is known as a **generic class** with a **type parameter E**. We bind **E** to a concrete type when creating an **ArrayList** object
- For example, the following statement creates an **ArrayList** object whose elements reference **String** objects

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



Pass the type argument String to type parameter E

[ArrayListTest.java](#)

Comparison between **Array** and **ArrayList**

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

ArrayLists from/to Arrays

- Create an ArrayList from an array of objects:

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

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

- Create an array of objects from an ArrayList:

```
var array1 = new String[list.size()]; // same as String[] array1 = ...  
list.toArray(array1);
```

ArrayArrayListConversion.java

Shuffling an **ArrayList**

```
import java.util.ArrayList;  
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList list = new ArrayList<Integer>(java.util.Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```

Enhanced for-loop

Can be used with any class that implements the **Iterable<T>** interface, such as ArrayList



local-variable type inference

Old way

```
for (int i = 0; i < circleArr.length; i++)  
    sum += circleArr[i].getArea();
```

```
for (int i = 0; i < circleArr.length; i++)  
    System.out.printf("%-30f%-15f\n",  
        circleArr[i].getRadius(), circleArr[i].getArea());
```

New way since Java 10

```
for (var c: circleArr) // enhanced for loop  
    sum += c.getArea();
```

```
for (var c: circleArr) // enhanced for loop  
    System.out.printf("%-30f%-15f\n",  
        c.getRadius(), c.getArea());
```

*Note: We will introduce **forEach** - a powerful loop operator for streaming data at the end of this course. It supports lazy parallel stream processing.*

TotalArea.java

Record as Data Carrier (Java 14 Preview Feature)

```
public final class Book {  
    private final String title;  
    private final ArrayList<String> authors;  
  
    public Book(String title, ArrayList<String> authors) {  
        this.title = title;  
        this.authors = authors;  
    }  
    public String title() {  
        return this.title;  
    }  
    public ArrayList<String> authors() {  
        return this.authors;  
    }  
    public boolean equals(Object o) {  
        if (o instanceof Book)  
            return title.equals(o.title) && authors.equals(o.authors);  
        else  
            return false;  
    }  
    public int hashCode() { ... }  
}
```



```
record Book (String title, ArrayList<String> authors) { }
```

A convenient feature!

- Java compiler automatically generates appropriate fields, constructor, getter methods. A record also implements equals(), hashCode() and toString() methods implicitly
- title and authors are the state description of the record
- We can define additional static fields and static/instance methods inside a record
- Semantics like data classes in Kotlin

Record – Multiple and Compact Constructors

```
record Book (String title, ArrayList<String> authors) {  
    public Book(String title) {  
        this(title, new ArrayList<String>());  
    }  
    public Book { // compact constructor  
        if (title == null || title.length() == 0)  
            throw new IllegalArgumentException();  
        if (authors.isEmpty())  
            authors.add("Anonymous");  
    }  
    public String title() { // redefine getter  
        return "The title of this book is " + this.title;  
    }  
    ...  
}
```

<https://www.journaldev.com/39987/java-records-class>

- A Record allows defining additional constructors but they cannot have the same signature as the **canonical constructor**:
 Book (String title, ArrayList<String> authors)
- A Record implicitly extends java.lang.Record
- A Record can implement multiple interfaces
- The **compact constructor** cannot have parameters
- It is not generated as a separate constructor but an extension of the canonical constructor
- It is executed at the entrance of the canonical constructor usually for parameter validation
- We can redefine the getter methods

Immutable Objects and Classes

- **Immutable object**: An object whose content cannot be changed once it is created
- **Immutable class**: The class of an immutable object
- In our example, if the Circle class has only private variables and no set methods in the Circle class, it is **immutable** because radius is private and cannot be changed without a mutator (set) method
- In general, a class that contains only private variables and get methods is not necessarily **immutable**. For example, Student contains only private variables and get methods, but it is **mutable**

Mutable Student Class

*assessor method returns
a mutable object!*

```
public class BirthDate {  
    private int year;  
    private int month;  
    private int day;  
  
    public void setYear(int newYear) {  
        year = newYear;  
    }  
  
    public BirthDate(int newYear,  
        int newMonth, int newDay) {  
        year = newYear;  
        month = newMonth;  
        day = newDay;  
    }  
}
```

```
public class Student {  
    private BirthDate birthDate;  
    public Student(int year, int month, int day) {  
        birthDate = new BirthDate(year, month, day);  
    }  
    public BirthDate getBirthDate() {  
        return birthDate;  
    }  
}
```

```
public class StudentMutableTest {  
    public static void main(String[] args) {  
        Student student = new Student(1980, 5, 3);  
        BirthDate date = student.getBirthDate();  
        date.setYear(2010); // birth year changed!  
    }  
}
```

What Classes are Immutable?

An **immutable class** must mark all variables private/final and provide no **mutator** (i.e., set) methods and no **accessor** (i.e., get) methods that would return a reference to a **mutable data field object**



Terminologies Clarification

- Does the term “Java method” implies it is an instance method?
 - No. There is no implication in official Java specification and documentation or popular online discussion forums that the term always implies an instance method.
- Is “default constructor” the same as “no-arg constructor”?
 - No. Default constructor refers to the no-arg constructor generated by Java compiler. Unlike default constructor, a no-arg constructor can have a non-empty body.
- Are “static methods” the same as “class methods”?
 - Yes. These two terms can be used interchangeably. The term “static methods” is more often found in Java literature.
- May a class be declared private or protected?
 - Yes if the class is defined inside another class. We see such use in the DefaultValueTest example and will see more later.

UNIT TESTING & JUNIT

Facts of a Real Enterprise Project

- Eclipse is known for its high reliability
- Defect density of Eclipse 3.0
 - ❑ Scariest defects: 30 per million LOC ← Lines of code
 - ❑ Scary: 160 per million LOC
 - ❑ Troubling: 480 per million LOC
 - ❑ Of concern: 6,000 per million LOC

What is unit testing?

- A unit typically refers to a method in a Java class.
- A unit test case is a test execution of a Java method.
- A test suite is a careful selected set of unit test cases.

What is JUnit ?

- JUnit is a regression testing framework written by Kent Beck and Enrich Gamma.
- Its latest version is 5, packaged as an IntelliJ plug-in.
- It is a de facto framework used by Java practitioners to test their code.



Kent Beck



Enrich Gamma

Requirement of a JUnit Test Case

@Test // annotates that the method following is a test

```
void getArea() {  
    var c = new Circle(10);  
    assertEquals(100*Math.PI, c.getArea(), 0.000001, "getArea() fails");  
}
```

expected value actual value delta error message

check actual
result against
expected result

- Test runs automatically with little human participation.
- Results are either pass or fail.
- Results are reproducible and consistent.

Putting it in IntelliJ

```
class CircleTest {  
    static final double epsilon = 0.00001;  
    @Test  
    void getArea() {  
        var c = new Circle(10);  
        assertEquals(100*Math.PI, c.getArea(), epsilon, "getArea() fails");  
    }  
    @Test  
    void getRadius() {  
        var c = new Circle(10);  
        assertEquals(10, c.getRadius(), epsilon, "getRadius() fails");  
    }  
}
```

CircleTest.java

BeforeEach and AfterEach test case

@BeforeEach

void setUp() {

// create objects and common variables for your tests here

// setUp() is called before EACH test case is run

}

@AfterEach

void tearDown() {

// put code here to reset or release objects, e.g., p1=null;

// to garbage collect unused objects or resources

// tearDown() is called after EACH test case is run

}

How tests in a Class are executed?

1. **setUp()**

2. ***Test 1***

3. **tearDown()**

4. **setUp()**

5. ***Test 2***

6. **tearDown()**

... until all test cases are executed

→ *Each test is independent of the previous ones*

Does expected = actual?

- To compare any variables of the 8 Java primitive types (i.e., boolean, byte, short, int, long, char, float, double) use:

assertEquals(expected, actual, delta)

- First argument (optional) is a string printed when the assertion fails:

assertEquals(expected, actual, delta, message)

Comparisons for Primitive Types

- `assertEquals(int expected, int actual)`
- `assertEquals(boolean expected, boolean actual)`
- `assertEquals(byte expected, byte actual)`
- `assertEquals(char expected, char actual)`
- `assertEquals(short expected, short actual)`
- `assertEquals(long expected, long actual)`
- `assertEquals(double expected, double actual, double delta)`
- `assertEquals(float expected, float actual, float delta)`
- Each of them supports a variant method by specifying a failure message string as the first formal parameter.

More Assertions

- `assertSame(expected, actual)`
- `assertSame(expected, actual, message)`
- `assertTrue(condition)`
- `assertTrue(condition, message)`
- `fail()`
- `fail(message)`
- `failNotEquals(expected, actual, message)`
- `failNotSame(expected, actual, message)`
- `failSame(message)`
- `assertFalse(condition)`
- `assertFalse(condition, message)`
- `assertNotNull(object)`
- `assertNotNull(object, message)`
- `assertNotSame(expected, actual)`
- `assertNotSame(expected, actual, message)`
- `assertNull(object)`
- `assertNull(object, message)`

API Documentation (JUnit 5) <https://junit.org/junit5/docs/current/api/index.html?overview-summary.html>

Test Creation (JUnit 5)

■ Create a JUnit test

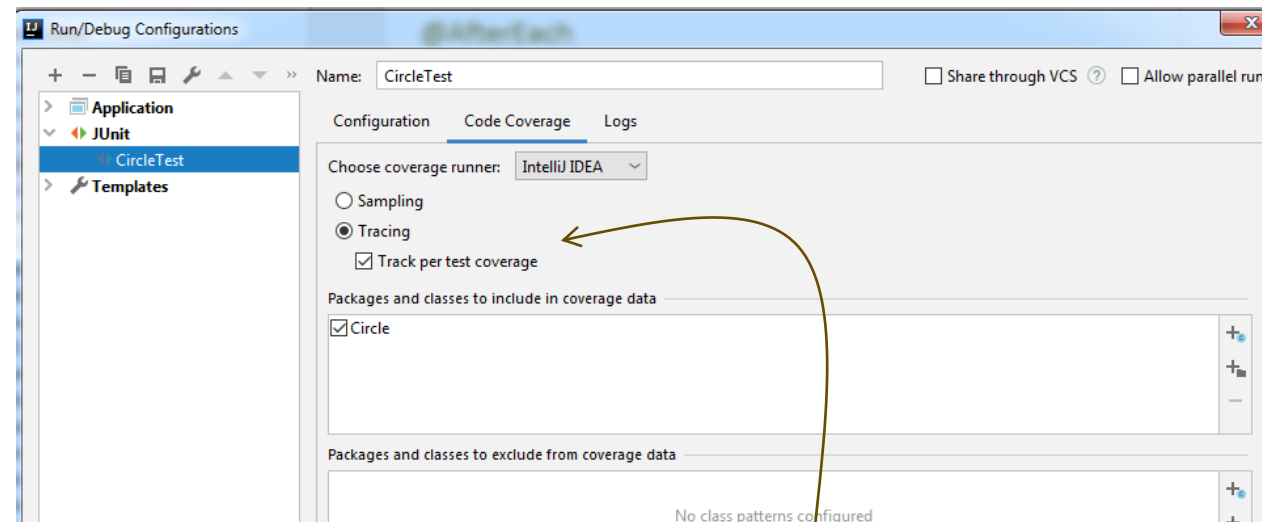
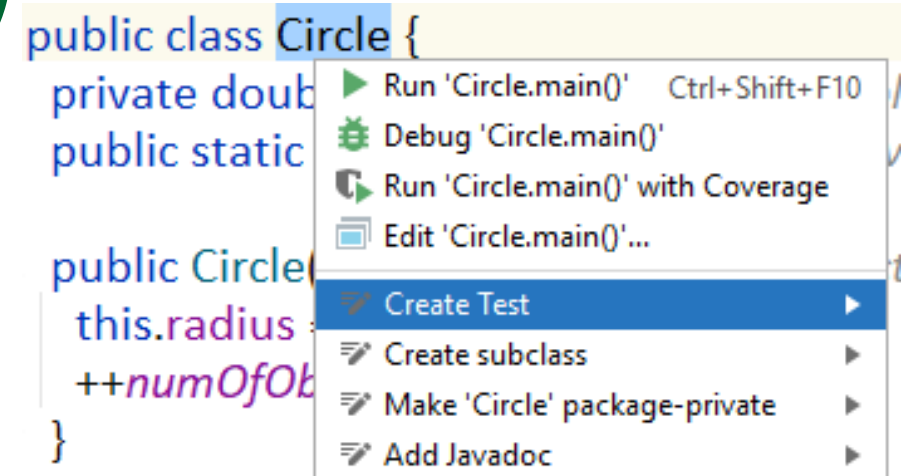
- ❑ Right-click the class to be tested
- ❑ Show Context Actions -> Create Test
- ❑ <https://www.jetbrains.com/help/idea/create-tests.html>

■ Configure a JUnit test

- ❑ Run -> Edit Configurations ...

■ Prepare for testing

- ❑ <https://www.jetbrains.com/help/idea/testing.html>
- ❑ https://datastork.io/intellij_setup_branch_code_coverage/



Enable branch coverage

Dynamic Tests

```
class DynamicDemoTest {
```

```
    @TestFactory
```

```
    Stream<DynamicTest> test() {
```

```
        // Generates tests for the first 10 even integers.
```

```
        return IntStream
```

```
            .iterate(0, n->n+2)
```

```
            .limit(10)
```

```
            .mapToObj(n ->
```

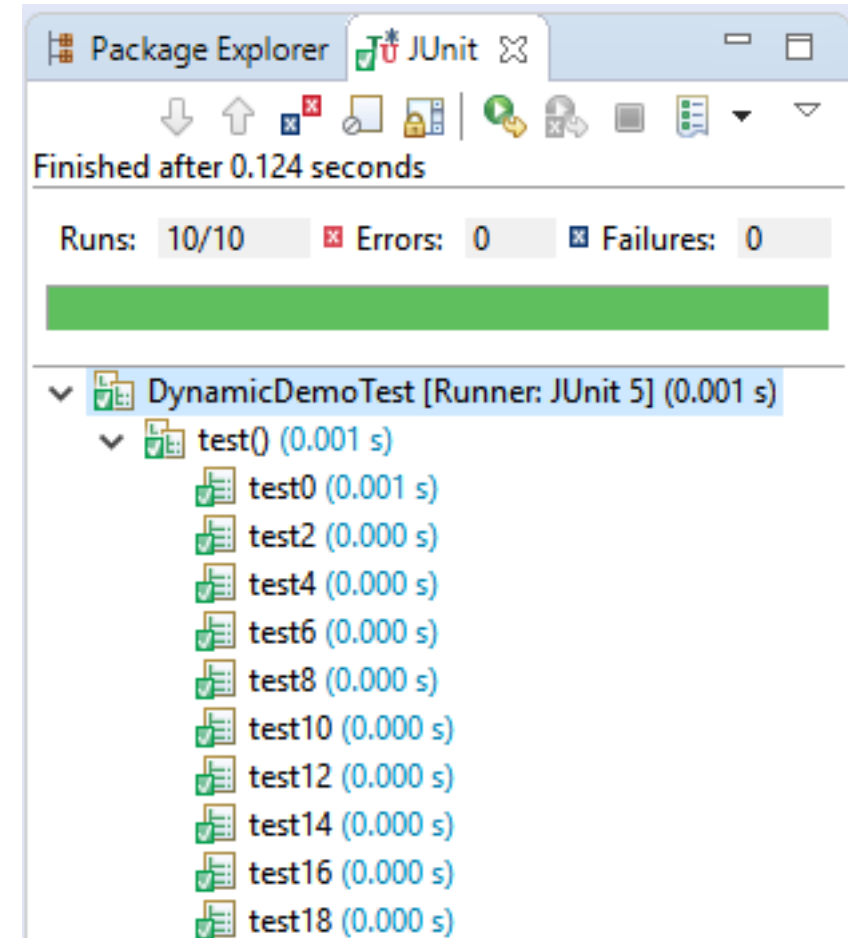
```
                DynamicTest.dynamicTest("test"+n, () -> assertTrue(n%2 ==0)));
```

```
    }
```

```
}
```

specify the test scripts to be generated

DynamicDemoTest





RECOMMENDED AFTER-CLASS PRACTICES

I am compiled from a .java file.

class

My instance variable values can be different from my buddy's values.

I behave like a template.

I like to do stuff.

I can have many methods.

I represent 'state'.

I have behaviors.

I am located in objects.

I live on the heap.

I am used to create object instances.

My state can change.

I declare methods.

I can change at runtime.

Who am I?

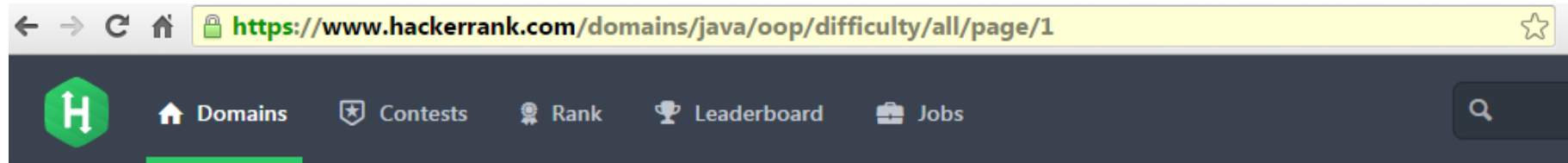
extracted from Head First Java

Note: both classes and objects are said to have state and behavior. They're defined in the class, but the object is also said to 'have' them. Right now, we don't care where they technically live.

I am compiled from a .java file.	class
My instance variable values can be different from my buddy's values.	object
I behave like a template.	class
I like to do stuff.	object, method
I can have many methods.	class, object
I represent 'state'.	instance variable
I have behaviors.	object, class
I am located in objects.	method, instance variable
I live on the heap.	object
I am used to create object instances.	class
My state can change.	object, instance variable
I declare methods.	class
I can change at runtime.	object, instance variable

Who am I?

More Practice - I



All Domains > Java > Object Oriented Programming

<https://www.hackerrank.com/domains/java/oop/difficulty/all/page/1>

Subdomains

Introduction

Strings

BigNumber

Data Structures

Object Oriented
Programming

Exception Handling

Advanced

Object Oriented Programming Challenges



Easy

Medium

Hard

Java Inheritance I

Success Rate: 98.64% Max Score: 5 Difficulty: Easy



Solve Challenge

Java Inheritance II

Success Rate: 97.28% Max Score: 10 Difficulty: Easy



Solve Challenge

Java Abstract Class

Success Rate: 98.72% Max Score: 10 Difficulty: Easy



Solve Challenge

More Practice - II

[Login](#)[Forgot Password](#)[PRACTICE](#) [COMPETE](#) [DISCUSS](#) [COMMUNITY](#) [HELP](#) [ABOUT](#)[Home](#) » All Submissions

ALL SUBMISSIONS

Language : Result : Year : User : Problem : Contest :

ID	Date/Time	User	Problem Code	Contest Code	Result	Time	Mem	Language	Solution
10860019	7 min ago	gurpreetgbit	CLEANUP	EASY	✓	0.17	1340.5M	JAVA	<input type="button" value="View"/>
10859962	19 min ago	pratik938	SNAPE	SCHOOL	✓	0.16	1341M	JAVA	<input type="button" value="View"/>
10859949	22 min ago	jay_11	LUCKFOUR	SCHOOL	✓ 100 [100pts]	0.93	1341M	JAVA	<input type="button" value="View"/>

<https://www.codechef.com/submissions>