NYU

Lecture 14

Objects and Classes

OO Programming Concepts

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

Objects

- An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

A class template

Class name: Circle

Data Fields:
radius is …

Methods:
getArea()

Three object of the Circle class

Circle Object 1

Data Fields:
radius is 10

Circle Object 2

Data Fields:
radius is 34

Circle Object 3

Data Fields:
radius is 113

Classes

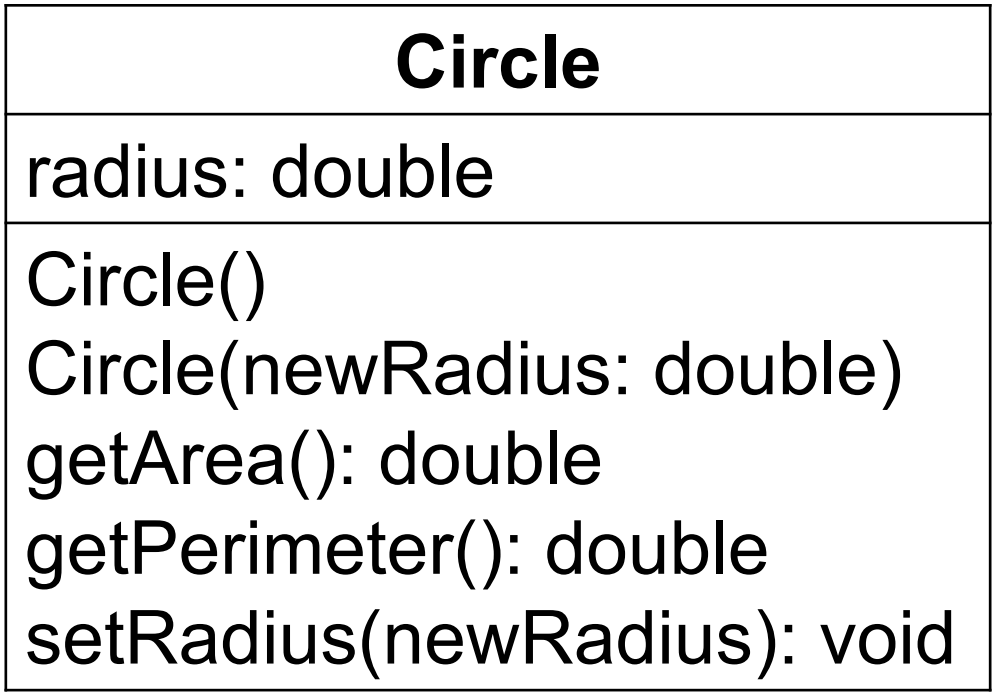
- ***Classes*** are constructs that define objects of the same type.
- A Java class uses variables to define **data fields** and **methods** to define behaviors.
- Additionally, a class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class.

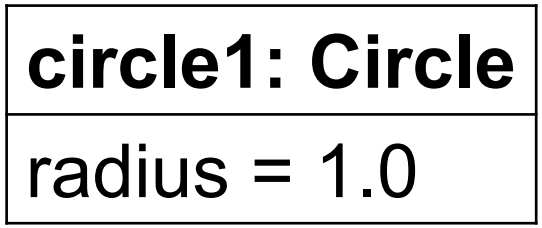
```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** 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 * 3.14159;
  }
}
```

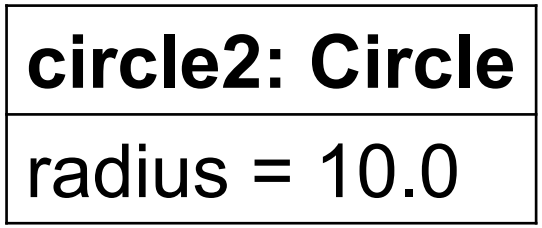
Data field

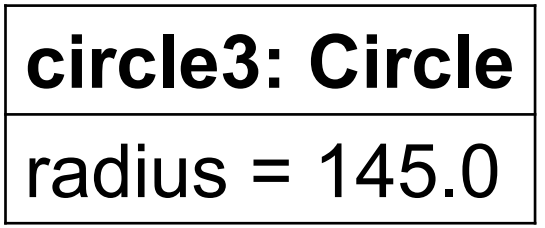
Constructors

Method


Circle
radius: double
Circle()
Circle(newRadius: double)
getArea(): double
getPerimeter(): double
setRadius(newRadius): void


circle1: Circle
radius = 1.0


circle2: Circle
radius = 10.0


circle3: Circle
radius = 145.0

Constructors

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

```
Circle() {
}

Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors

- A constructor with no parameters is referred to as a **no-arg** constructor.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created.
- Constructors play the role of initializing objects.

Creating Objects Using Constructors

- `new ClassName();`

- Example:

- `new Circle();`
- `new Circle(5.0);`

Default Constructor

- A class may be defined **without** constructors.
- In this case, a no-arg constructor with an empty body is **implicitly defined** in the class.
- This constructor, called *a* ***default constructor***
- Is provided automatically *only if no constructors are explicitly defined in the class*.

Declaring Object Reference Variables

- To reference an object, assign the object to a reference variable.
- To declare a reference variable, use the syntax:
- `ClassName objectRefVar;`
- Example:
- `Circle myCircle;`

Declaring/Creating Objects in a Single Step

- `ClassName objectRefVar = new ClassName();`

- Example:
- `Circle myCircle = new Circle();`

Assign object reference Create an object

Accessing Object's Members

- Referencing the object's data:
  - `objectRefVar.data`
  - `myCircle.radius`

- Invoking the object's method:
  - `objectRefVar.methodName(arguments)`
  - `myCircle.getArea()`

Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

Declare myCircle

myCircle | no value

Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

Create a circle

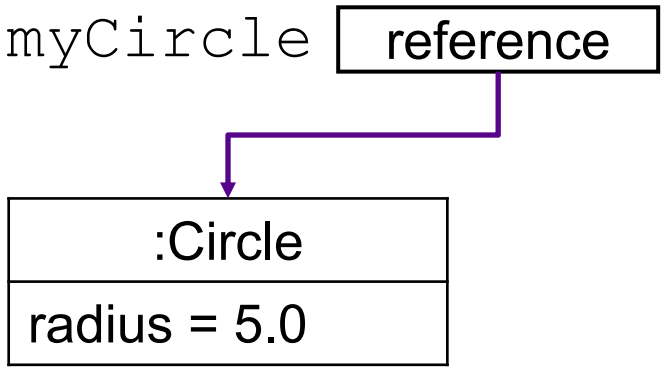
myCircle | no value

:Circle
radius = 5.0

Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

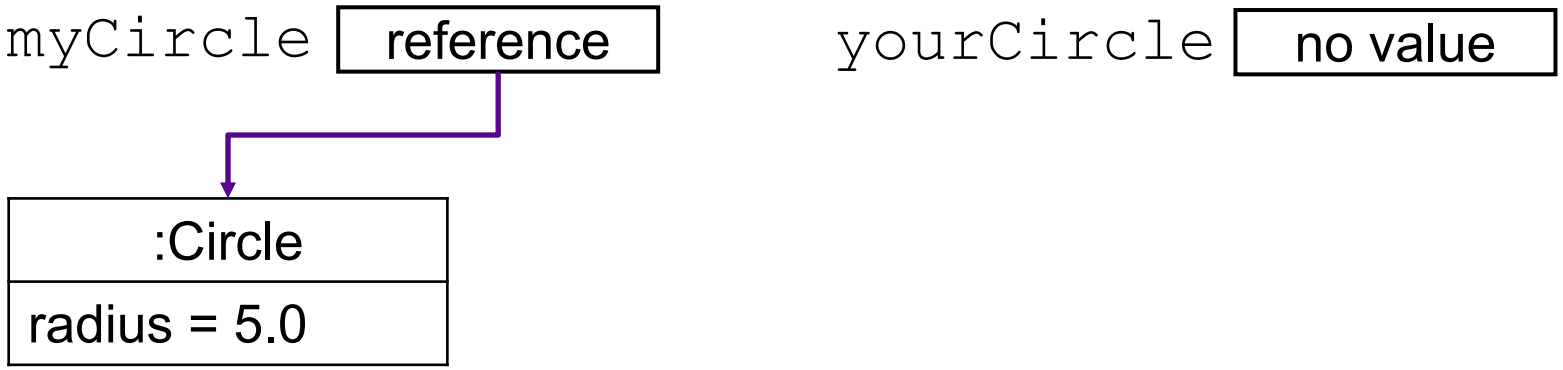
Assign object reference to `myCircle`



Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

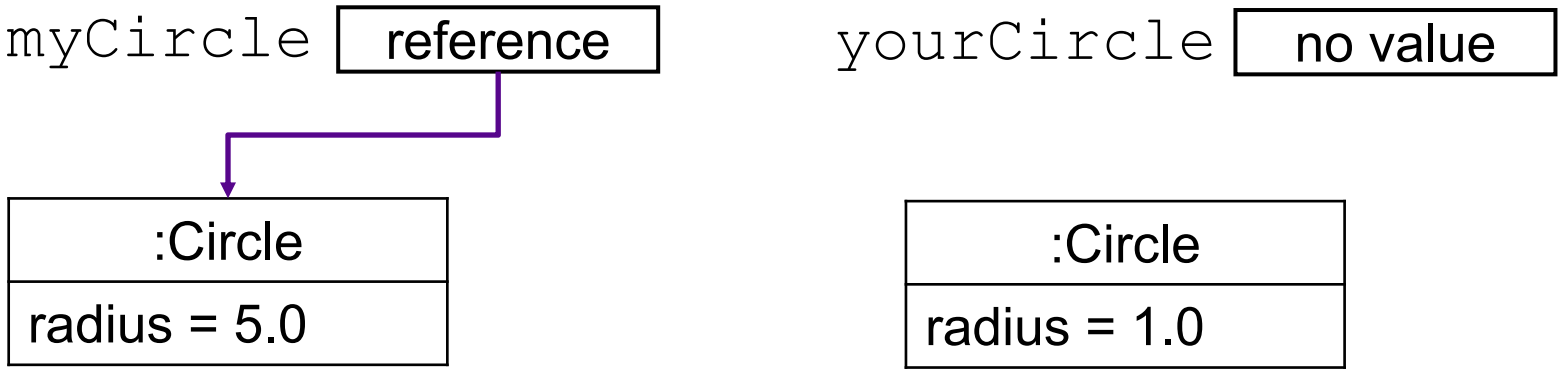
Declare `yourCircle`



Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

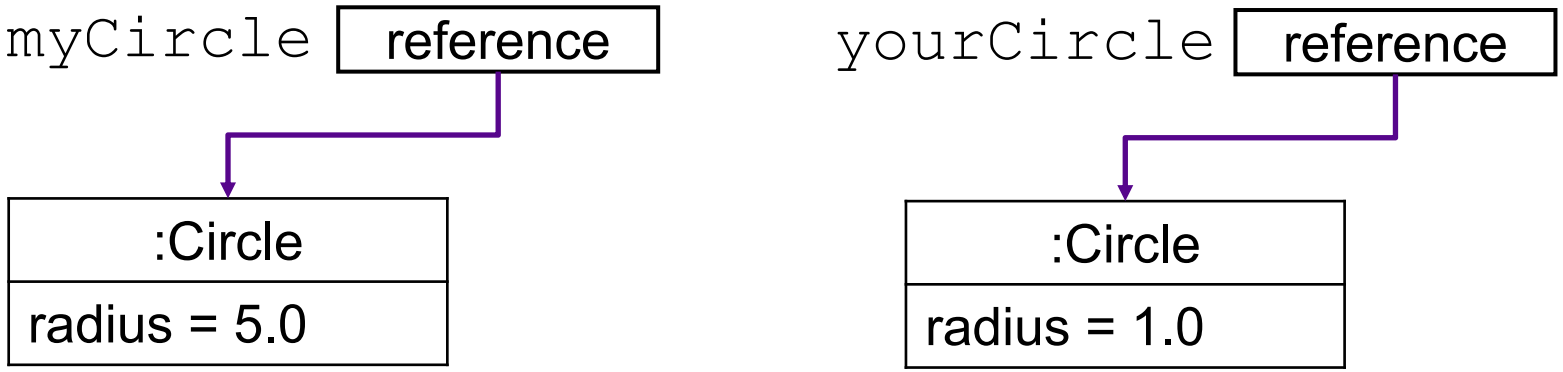
Create a new `Circle` object



Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

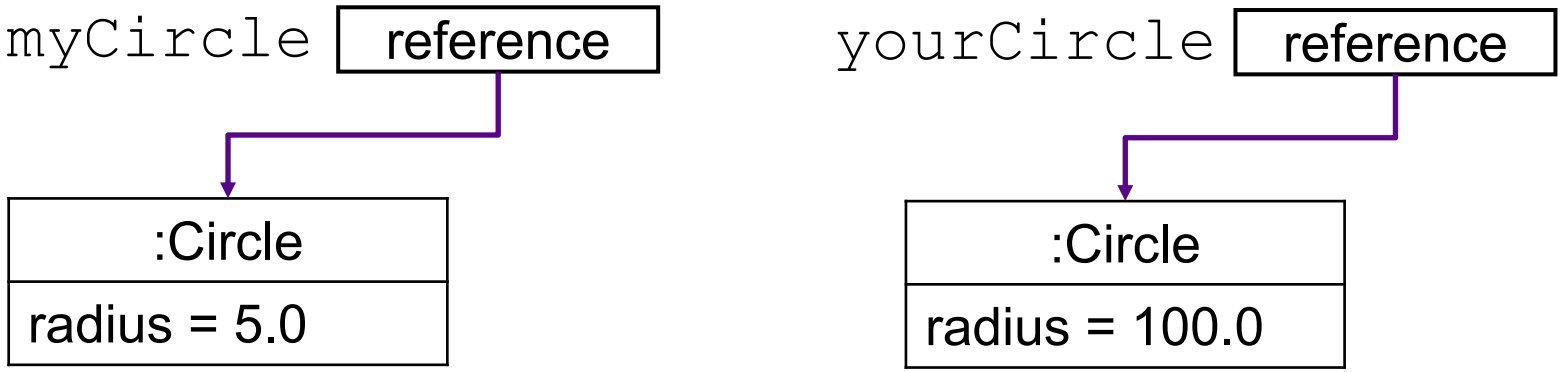
Assign object reference to `yourCircle`



Trace Code

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

Change radius in `yourCircle`



Caution

- Recall that you use to invoke a method in the Math class `Math.methodName(arguments)`
  - `Math.pow(3, 2.5)`

- Can you invoke getArea() using SimpleCircle.getArea()?
  - The answer is no.

- All the methods used before until now are static methods, which are defined using the **static** keyword.
- `getArea()` is non-static. It must be invoked from an object using `objectRefVar.methodName(arguments)`
  - `myCircle.getArea()`

- More explanations will be given in the section on “Static Variables, Constants, and Methods.”

Reference Data Fields

- The data fields can be of reference types. For example, the following Student class contains a data field name of the String type.

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

Default Value for a Data Field

- If a data field of a reference type does not reference any object, the data field holds a special literal value, **null**.
- The default value of a data field is 0 for a numeric type, false for a boolean type, and '\u0000' for a char type.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " +
  student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

Example

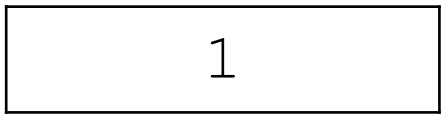
- Java assigns **no default value** to a local variable inside a method.

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

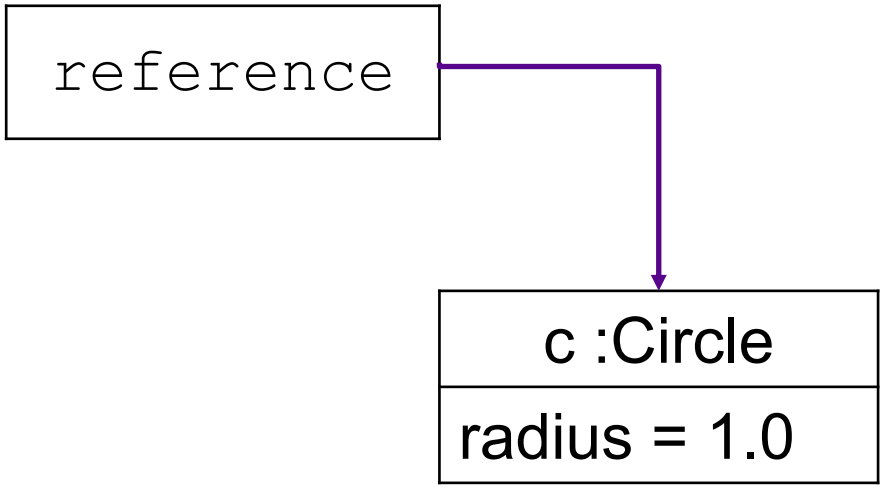
Compile error: variable not initialized

Primitive Data Types vs Object Types

Primitive type `int i = 1`



Object type `Circle c`

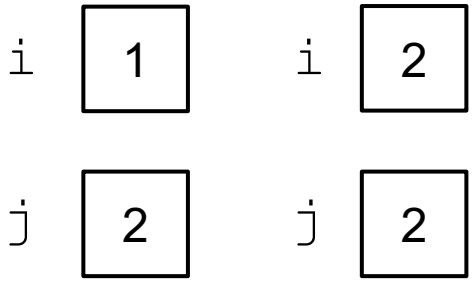


Copying Variables

Primitive type assignment

```
i = j
```

Before

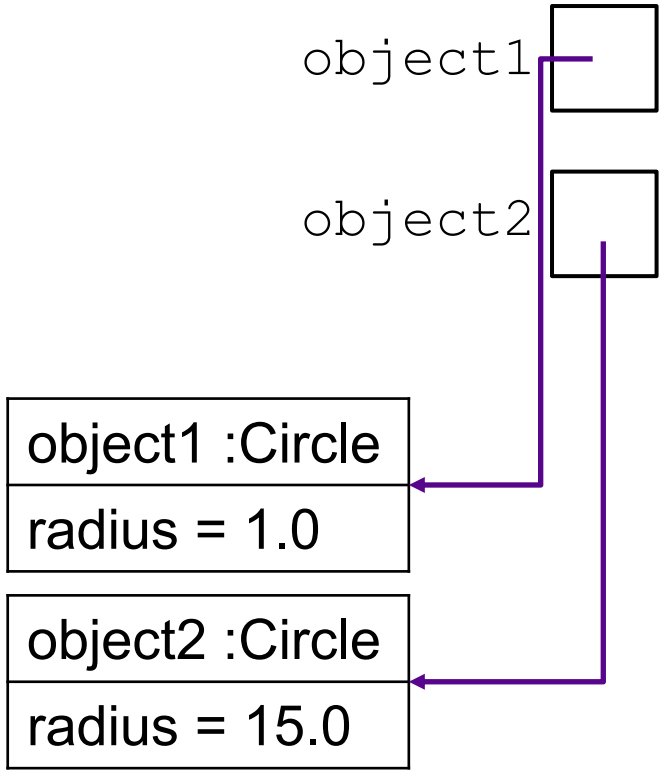


After

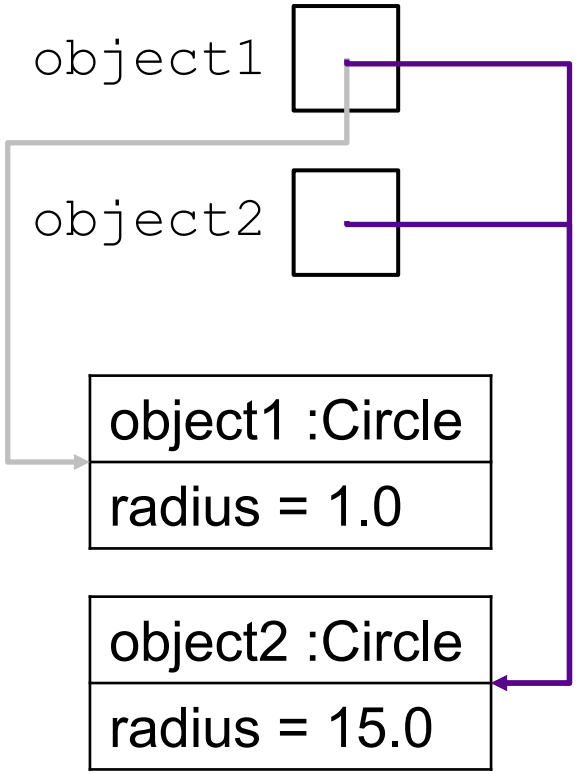
Object type assignement

```
object1 = object2
```

Before



After



Garbage Collection

- After the assignment statement
  `object1 = object2`
- `object1` points to the same object referenced by `object2`.

- The object previously referenced by `object1` is no longer referenced.

- This object is known as **garbage**.

- Garbage is automatically collected by JVM.

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

The Date Class

- Java provides a system-independent encapsulation of date and time in the `java.util.Date` class.
- You can use the Date class to create an instance for the current date and time and use its `toString` method to return the date and time as a string.

java.util.Date	
Date()	Constructs a Date for the current time
Date(elapseTime: long)	Constructs a Date for a given time
toString(): String	Returns a string representing the date
getTime(): long	Returns the number of ms since 1/1/1970 GMT
setTime(elapseTime: long): void	Sets the number of ms since 1/1/1970 GMT

The Date Class Example

- For example, the following code
  - `java.util.Date date = new java.util.Date();`
  - `System.out.println(date.toString());`

- displays a string like Sun Mar 09 13:50:19 EST 2018.

The Random Class

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

java.util.Random	
Random()	Constructs a Random with the current time as seed
Random(seed: long)	Constructs a Random 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
nextFloat(): float	Returns a random float value
nextBoolean(): boolean	Returns a random boolean value

The Random Class Example

- If two Random objects have the same seed, they will generate identical sequences of numbers.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

From `random1`: 734 660 210 581 128 202 549 564 459 961
From `random2`: 734 660 210 581 128 202 549 564 459 961

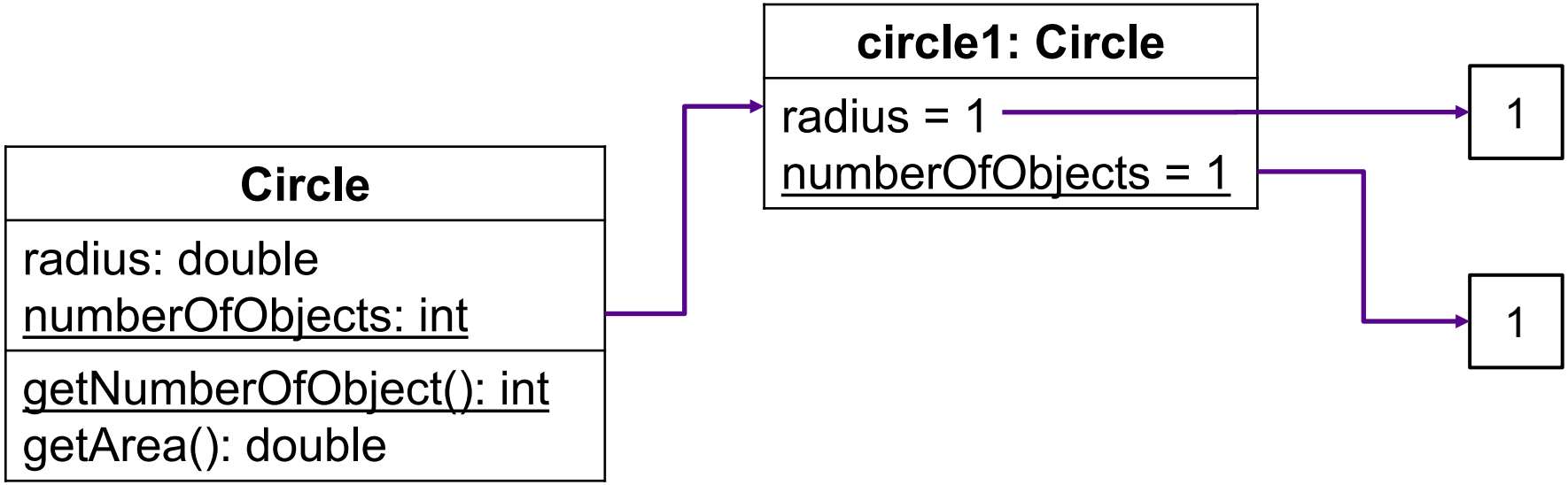
Instance Variables, and Methods

- Instance
  - Instance variables belong to a specific instance.
  - Instance methods are invoked by an instance of the class.

- Static
  - Static variables are shared by all the instances of the class.
  - Static methods are not tied to a specific object.
  - Static constants are final variables shared by all the instances of the class.
  - To declare static variables, constants, and methods, use the static modifier.

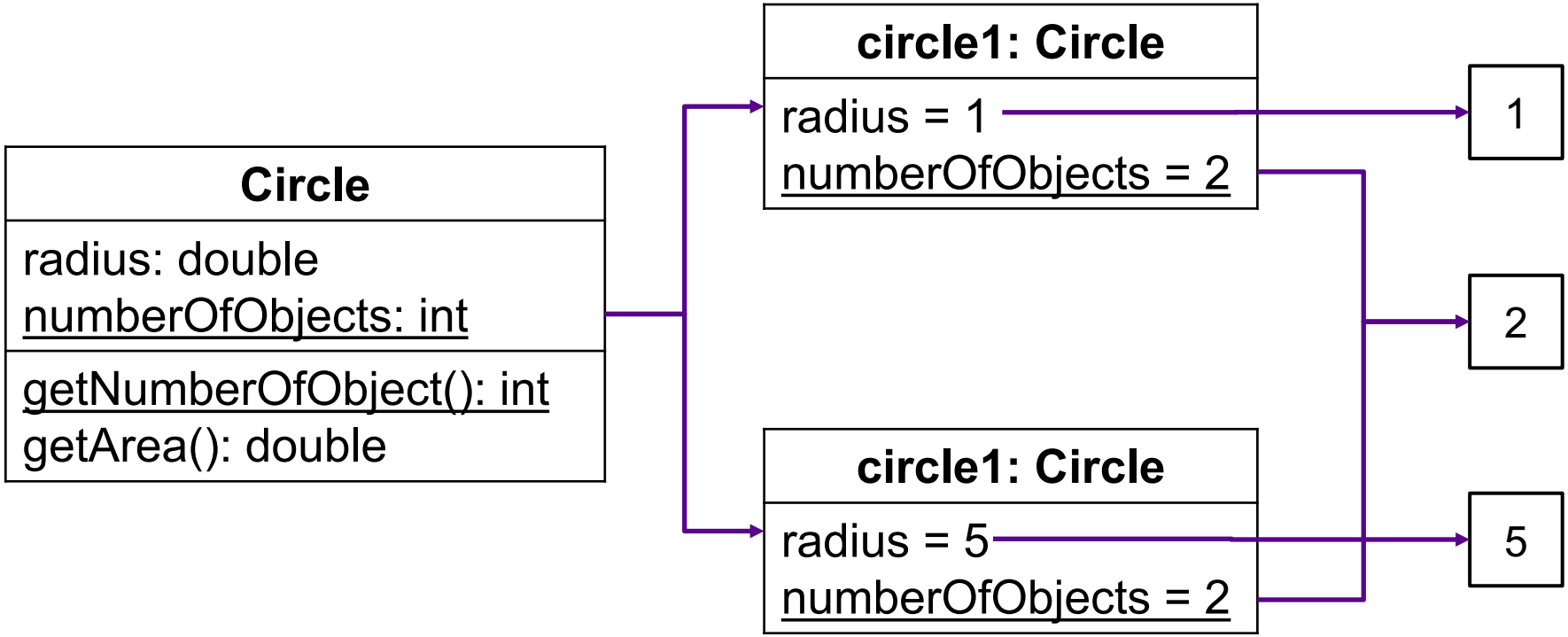
Static Variables, Constants, and Methods

- UML Notation: underline static variable of methods



Static Variables, Constants, and Methods

- UML Notation: underline static variable of methods



Visibility Modifiers and Accessor/Mutator Methods

- By default, the class, variable, or method can be accessed by any class in the same package.

- **`public`**
  The class, data, or method is visible to any class in any package.

- **`private`**
  The data or methods can be accessed only by the declaring class.

- The **get** and **set** methods are used to read and modify private properties.

Example

- 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.

```
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()
  }
}
```

Example

- 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.

```
package p1;

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

  public void m1() {
  }
  void m2() {
  }
  private void 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

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

```
package p1;

class C1 {
   ...
}
```

```
package p1;

public class C2 {
   //can access C1
}
```

```
package p2;

public class C2 {
   //cannot access C1
   //can access C2
}
```

- An object can access in its own private members

```
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;
  }
}
```

- Any other object cannot access 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());
  }
}
```

Why Data Fields Should Be private?

- To protect data.
- To make code easy to maintain.

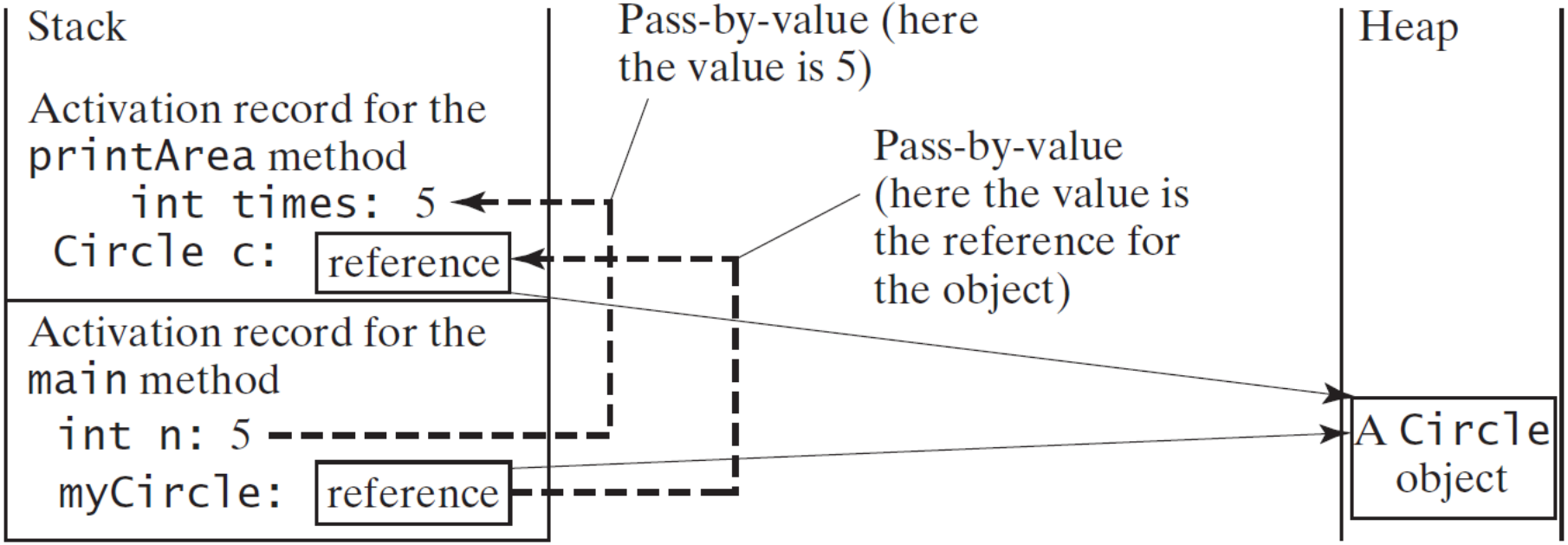
Example of Data Field Encapsulation

- UML Notation:
  - + sign indicates **public** modifier
  - - sign indicates **private** modifier

Circle	
- radius: double	The radius of the circle (default: 1)
- numberOfObjects: int	The number of created objects
+ Circle()	Constructs a default circle
+ Circle(radius: double)	Constructs a circle with a specified radius
+ getRadius(): double	Returns the radius of this circle
+ setRadius(radius: double): void	Sets a new radius for this circle
+ getArea(): double	Returns the area of this circle
+ getNumberOfObjects(): int	Returns the number of circle objects created

Passing Objects to Methods

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

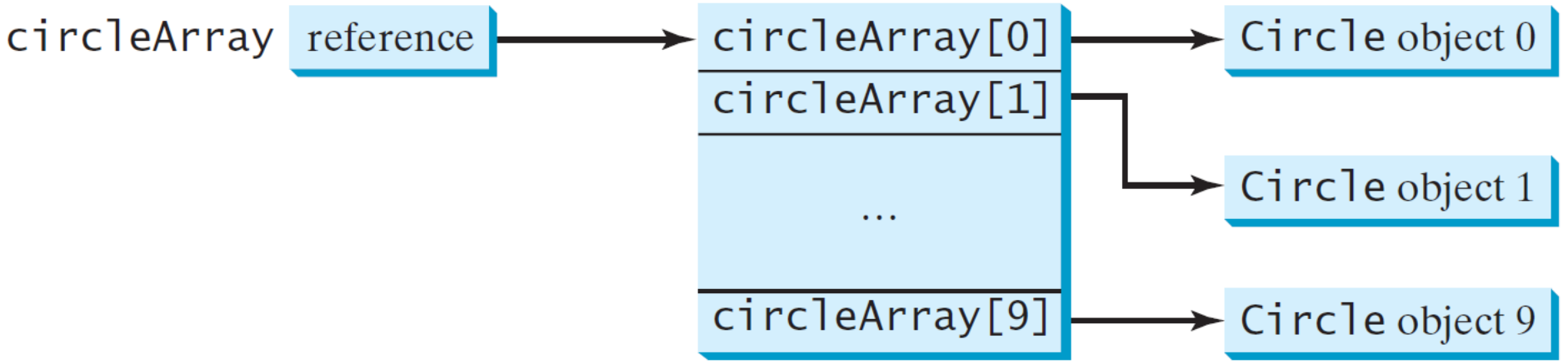


Array of Objects

- `Circle[] circleArray = new Circle[10];`
- An array of objects is actually an array of reference variables.
- So invoking `circleArray[1].getArea()` involves two levels of referencing.
- `circleArray` references to the entire array.
- `circleArray[1]` references to a `Circle` object.

Array of Objects

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



Immutable Objects and Classes

- If the contents of an object cannot be changed once the object is created, the object is called an **immutable object** and its class is called an **immutable class**.
- If you delete the set method in the Circle, the class would be immutable because radius is private and cannot be changed without a set method.

- A class with all private data fields and without mutators is not **necessarily immutable**.

Example

```
public class Student {
  private int id;
  private BirthDate birthDate;

  public Student(int ssn,
      int year, int month, int day) {
    id = ssn;
    birthDate = new BirthDate(year, month, day);
  }

  public int getId() {
    return id;
  }

  public BirthDate getBirthDate() {
    return birthDate;
  }
}
```

Example

```
public class BirthDate {
  private int year;
  private int month;
  private int day;

  public BirthDate(int newYear,
      int newMonth, int newDay) {
    year = newYear;
    month = newMonth;
    day = newDay;
  }

  public void setYear(int newYear) {
    year = newYear;
  }
}
```

Example

```
public class Test {
  public static void main(String[] args) {
    Student student =
              new Student(111223333, 1970, 5, 3);
    BirthDate date = student.getBirthDate();
    date.setYear(2010);
   // Now the student birth year is changed!
  }
}
```

What Class is Immutable?

- For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.

Scope of Variables

- The scope of instance and static variables **is the entire class**.
- They can be declared anywhere inside a class.

- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- A local variable must be initialized explicitly before it can be used.

The this Keyword

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

- One common use of the this keyword is reference a class's hidden data fields.
- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

Reference the Hidden Data Fields

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

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

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

Suppose that `f1` and `f2` are two objects of `F`.

```
F f1 = new F();
F f2 = new F();
```

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

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

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

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

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

this is used to invoke another constructor

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