

The slide features five circles of varying shades of light purple. Two circles are solid, and three are hollow with a thin outline. They are arranged in a loose, abstract pattern around the central text.

# Classes & Objects

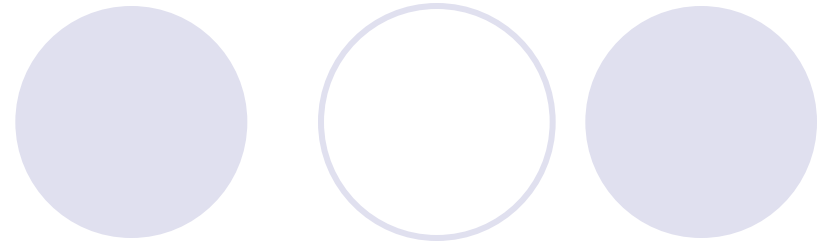
*By Pakita Shamoi*

# Objects, classes



- “Class” refers to a blueprint. It defines the **variables** and **methods** the objects support
- “Object” is an instance of a class. Each object has a class which defines its **data** and **behavior**

# Class Members



- **A class can have:**

- ***fields***: data variables which determine the **status** of the class or an object
- ***methods***: executable code of the class built from statements. It allows us to manipulate/change the status of an object or access the value of the data member. Determines **behavior** of the object.

# Class example

```
class Pencil {  
    public String color = "red";  
    public int length;  
    public double diameter;  
    public void setColor (String newColor)  
    {  
        color = newColor;  
    }  
}
```

# Fields – Declaration

- a type name followed by the field name
- field declarations can be preceded by different modifiers
  - access control modifiers
  - static
  - final

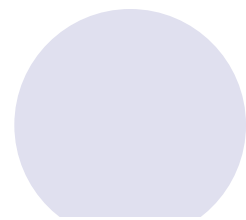
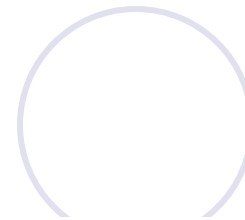
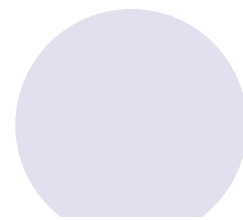
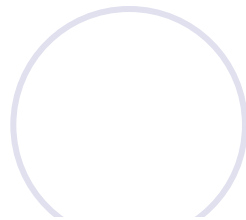
```
public String color = "red";
```

# Access control modifiers



- *private*: private members are accessible only in the class itself
- *package*: package members are accessible in classes in the same package and the class itself
- *protected*: protected members are accessible in classes in the same package, in subclasses of the class, and in the class itself
- *public*: public members are accessible anywhere the class is accessible

You will often use *private* and *public*.



<b>Access Modifiers</b>	<i>Same Class</i>	<i>Same Package</i>	<i>Subclass</i>	<i>Other packages</i>
<b>public</b>	Y	Y	Y	Y
<b>protected</b>	Y	Y	Y	N
<b>no access modifier</b>	Y	Y	N	N
<b>private</b>	Y	N	N	N

## Pencil.java

```
public class Pencil {  
    public String color = "red";  
    public int length;  
    public double diameter;  
    private double price;  
  
    public static long numberOfPencils = 0;  
    public void setPrice (float newPrice) {  
        price = newPrice;  
    }  
}
```

## CreatePencil.java

```
public class CreatePencil {  
    public static void main (String args[]){  
        Pencil p1 = new Pencil();  
        p1.price = 0.5f;  
    }  
}
```

```
%> javac Pencil.java
```

```
%> javac CreatePencil.java
```

```
CreatePencil.java:4: price has private access in Pencil  
    p1.price = 0.5f;  
    ^
```




# Static



- Only one copy of the static field exists, shared by all objects of this class
- Can be accessed directly in the class itself
- Access from outside the class must be preceded by the class name as follows

```
System.out.println(Pencil.ID);
```

- From outside the class, non-static fields must be accessed through an **object reference**



```
public class CreatePencil {  
    public static void main (String args[]) {  
        Pencil p1 = new Pencil();  
        Pencil.numberOfPencils ++;  
        System.out.println(p1.numberOfPencils);  
        //Result? 1  
  
        Pencil p2 = new Pencil();  
        Pencil.numberOfPencils ++;  
        System.out.println(p2.numberOfPencils);  
        //Result? 2  
  
        System.out.println(p1.numberOfPencils);  
        //Result?  Again 2!  
    }  
}
```

Note: this code is only for the purpose of showing the usage of static fields. It has POOR design!

# Final



- once initialized, the value cannot be changed
- often be used to define named constants
- **static final** fields must be initialized when the **class** is initialized
- **non-static final** fields must be initialized when an **object** of the class is constructed

# Default values

- If a variable was not initialized, then a default initial value is assigned depending on its type

Type	Initial Value
boolean	false
char	'\u0000'
byte, short, int, long	0
float	+0.0f
double	+0.0
object reference	null

# Methods – Declaration

- Method declaration: two parts

1. **method header**

- consists of modifiers (optional), return type, method name, parameter list and a throws clause (optional)
- types of modifiers
  - *access control modifiers*
  - *abstract*
    - the method body is empty. E.g.  

```
abstract void sampleMethod( );
```
  - *static*
    - represent the whole class, not a specific object
    - can only access static fields and other static methods of the same class
  - *final*
    - cannot be overridden in subclasses

2. **method body**

# Methods – Invocation

- Method invocations

- invoked as operations on objects/classes using the dot ( . ) operator

**`reference.method(arguments)`**

- **static method:**

- Outside of the class: “`reference`” can either be the class name or an object reference belonging to the class
- Inside the class: “`reference`” can be omitted

- **non-static method:**

- “`reference`” must be an object reference

# Method - Overloading

- A class can have more than one method with the same name as long as they have different parameter list.

```
public class Pencil {  
    . . .  
    public void setPrice (double newPrice) {  
        price = newPrice;  
    }  
  
    public void setPrice (Pencil p) {  
        price = p.getPrice();  
    }  
}
```

- How does the compiler know which method you're invoking? — compares the number and type of the parameters and uses the matched one

# Methods – Parameter Values

- Parameters are always passed by value.

```
public void method1 (int a) {  
    a = 6;  
}
```

```
public void method2 ( ) {  
    int b = 3;  
    method1(b);    // now b = ?  
                  // b = 3  
}
```

- When the parameter is an object reference, it is the object reference, not the object itself, getting passed.



# Methods – Reference Values

```
public void method1 (Student a) {  
    a.setName("Aray");  
}  
public void method2 ( ) {  
    Student b = new Student();  
    b.setName("Adiya");  
    method1(b);    // now s is ?  
}
```

The value of *b* is not changed. But the name of *Student b* is changed, since it is equivalent to:

```
a = b;  
a.setName("Aray");
```

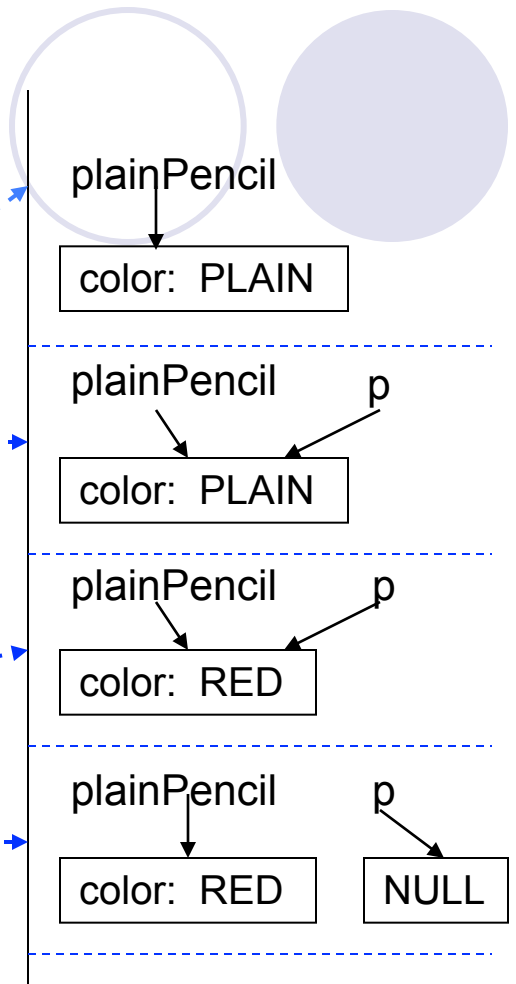
## another example: (parameter is an object reference)

```
class PassRef{
    public static void main(String[] args) {
        Pencil plainPencil = new Pencil("PLAIN");
        System.out.println("original color: " +
            plainPencil.color);

        paintRed(plainPencil);

        System.out.println("new color: " +
            plainPencil.color);
    }

    public static void paintRed(Pencil p) {
        p.color = "RED";
        p = null;
    }
}
```



- If you change any field of the object which the parameter refers to, the object is changed for every variable which holds a reference to this object

# The Main Method - Concept

- **main** method

- the system locates and runs the main method for a class when you run a program
- other methods get execution when called by the main method explicitly or implicitly
- must be **public**, **static** and **void**

# Modifiers of the classes

- A class can also have modifiers
  - **public**
    - publicly accessible
    - without this modifier, a class is only accessible within its own package
  - **abstract**
    - no objects of abstract classes can be created
    - all of its abstract methods must be implemented by its subclass; otherwise that subclass must be declared `abstract` also
    - will be discussed later
  - **final**
    - can not be subclassed
- Normally, a file can contain multiple classes, but **only one public** one. The file name and the public class name should be the same

# Object Creation

```
class Body {  
    private long idNum;  
    private String name  
= "empty";  
    private Body  
orbits;  
    private static long  
nextID = 0;  
}
```

Body sun = new Body( );

**define** a variable  
*sun* to refer to a  
Body object

**create** a new  
Body object

- An object is created by the **new** method
- The runtime system will allocate enough memory to store the new object
- If no enough space, the automatic garbage collector will reclaim space from other no longer used objects. If there is still no enough space, then an `OutOfMemoryError` exception will be thrown
- No need to delete explicitly

# Constructor

- Constructor is a way to initialize an object
- Can have any of the same access modifiers as class members
- A class can have multiple constructors as long as they have different parameter list. Constructors have **NO** return type.
- Constructors with no arguments are called *no-arg* constructors.
- If no constructor is provided explicitly by the programmer, then the language provides a *default no-arg* constructor which sets all the fields which has no initialization to be their default values. It has the same accessibility as its class.

# Constructor

```
public class Entry {  
    private String name; // name as characters  
    private String number; //phone number  
    public Entry(String person, String phone) {  
        name = person; // initialise name  
        number = phone; // initialise number  
    }  
}
```

Then Entry might be set up as follows:

```
Entry newEntry = new Entry("Ainur", "+77013467222");
```

## Sample Class and Constructors

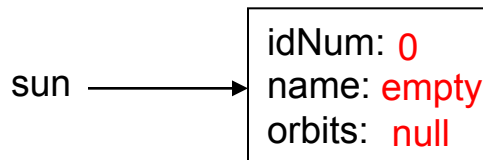
```
class Body {
    private long idNum;
    private String name= "empty";
    private Body orbits;
    private static long nextID = 0;

    Body( ) {
        idNum = nextID++;
    }

    Body(String bodyName, Body orbitsAround) {
        this( );
        name = bodyName;
        orbits = orbitsAround;
    }
}
```

Assume no any Body object is constructed before:

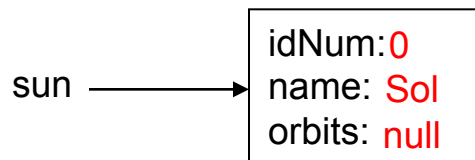
```
Body sun = new Body( );
```



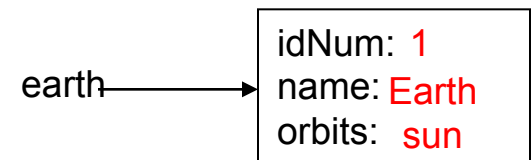
nextID = 1

Assume no any Body object is constructed before:

```
Body sun = new Body("Sol", null);
Body earth = new Body("Earth", sun);
```



nextID = 1



nextID = 2



# Usage of *this*

- Inside a constructor, you can use **this** to invoke another constructor in the same class. This is called *explicit constructor invocation*.
- It **MUST** be the first statement in the constructor body if exists.
- **this** can also be used as a reference of the current object.
- It **CANNOT** be used in a static method. *Why?*

# Usage of this

```
public class Point {  
    int x = 0;  
    int y = 0;  
    //constructor  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public Circle(){  
    radius = 1.0;  
    center = new Point(0,0);  
}  
public Circle(double r) {  
    radius = r;  
}  
public Circle(Point center, double r){  
    this(r);  
    this.center = center;  
}
```

## Example: usage of **this** as a reference of the current object

```
class Body {
    private long idNum;
    private String name;
    private Body orbits;
    private static long nextID = 0;
    private static LinkedList bodyList = new LinkedList();

    . . .

    Body(String name, Body orbits) {
        this.name = name;
        this.orbits = orbits;
    }

    . . .

    private void inQueue() {
        bodyList.add(this);
    }

    . . .
}
```

# Initialization block

- Initialization block
  - a block of statements **to initialize** the fields of the object
  - outside of any member or constructor declaration
  - they are executed **BEFORE** the body of the constructors!

## Without initialization block

```
class Body {  
    private long idNum;  
    private String name = "noNameYet";  
    private Body orbits;  
    private static long nextID = 0;  
  
    Body( ) {  
        idNum = nextID++;  
    }  
  
    Body(String bodyName, Body orbitsAround)  
    {  
        this( );  
        name = bodyName;  
        orbits = orbitsAround;  
    }  
}
```

=

## With initialization block

```
class Body {  
    private long idNum;  
    private String name = "noNameYet";  
    private Body orbits;  
    private static long nextID = 0;  
  
    {  
        idNum = nextID++;  
    }  
  
    Body(String bodyName, Body orbitsAround)  
    {  
        name = bodyName;  
        orbits = orbitsAround;  
    }  
}
```

# Static initialization block

- Resembles a non-static initialization block except that it is declared `static`, **can only refer to static members** and cannot throw any checked exceptions
- Gets executed when the class is first loaded

```
class Primes {  
    static int[] primes = new int[4];  
  
    static {  
        primes[0] = 2;  
        for(int i=1; i<primes.length; i++) {  
            primes[i] = nextPrime( );  
        }  
    }  
    //declaration of nextPrime( ) . . .  
}
```

# Packages

- Classes can be grouped in a collection called *package*
- Java's standard library consists of hierarchical packages, such as java.lang and java.util

<http://java.sun.com/j2se/1.4.2/docs/api>

- Main reason to use package is to **guarantee the uniqueness of class names** - classes with same names can be encapsulated in different packages

# Class importation (1)

- Two ways of accessing **PUBLIC** classes of another package
  - 1) explicitly give the full package name before the class name.

```
java.util.Date today = new java.util.Date( );
```
  - 2) import the package by using the **import** statement at the top of your source files (but below package statements).
    - to import a single class from the java.util package

```
import java.util.Date;  
Date today = new Date( );
```
    - to import all the public classes from the java.util package

```
import java.util.*;
```
    - \* is used to import classes at the current package level. It will **NOT** import classes in a sub-package.

# Class importation (2)

- What if you have a name conflict?

```
import java.util.*;
import java.sql.*;
Date today = new Date( ); //ERROR:java.util.Date
                           //or java.sql.Date?
```

- if you only need to refer to one of them, import that class explicitly

```
import java.util.*;
import java.sql.*;
import java.util.Date;
Date today = new Date( );           // java.util.Date
```

- if you need to refer to both of them, you have to use the full package name before the class name

```
import java.util.*;
import java.sql.*;
java.sql.Date today = new java.sql.Date( );
java.util.Date nextDay = new java.util.Date( );
```



See this code:

```
import java.lang.Math;

public class importTest {
    double x = sqrt(1.44);
}
```

Compile:

```
%> javac importTest.java
importTest.java:3: cannot find symbol
symbol   : method sqrt(double)
location: class importTest
double x = sqrt(1.44);
            ^
1 error
```



Remember, for the static members, you refer them as **className.memberName**, in our case it will be **Math.sqrt(1.44)**

# Static importation

- In J2SE 5.0, importation can also be applied on static fields and methods, not just classes. You can directly refer to them after the static importation.

- E.g. import all static fields and methods of the Math class

```
import static java.lang.Math.*;  
double x = PI;
```

- E.g. import a specific field or method

```
import static java.lang.Math.abs;  
double x = abs(-1.0);
```

- Any version before J2SE 5.0 does NOT have this feature!

# To emphasize on data encapsulation (1)

Let's see a sample class first

```
public class Body {
    public long idNum;
    public String name = "<unnamed>";
    public Body orbits = null;
    public static long nextID = 0;

    Body( ) {
        idNum = nextID++;
    }

    Body(String bodyName, Body orbitsAround) {
        this( );
        name = bodyName;
        orbits = orbitsAround;
    }
}
```

**Problem:** all the fields are exposed to change by everybody

# To emphasize on data encapsulation (2)

improvement on the previous sample class with data encapsulation

```
public class Body {  
    private long idNum;  
    private String name = "<unnamed>";  
    private Body orbits = null;  
    private static long nextID = 0;  
  
    Body( ) {  
        idNum = nextID++;  
    }  
  
    Body(String bodyName, Body orbitsAround) {  
        this( );  
        name = bodyName;  
        orbits = orbitsAround;  
    }  
}
```

**Problem:** but how can you access the fields?

# get / set methods

improvement on the previous sample class **with accessor methods**

```
public class Body {
    private long idNum;
    private String name = "<unnamed>";
    private Body orbits = null;
    private static long nextID = 0;

    Body( ) {
        idNum = nextID++; }

    Body(String bodyName, Body orbitsAround) {
        this( );
        name = bodyName;
        orbits = orbitsAround; }

    public long getID() {return idNum;}
    public String getName() {return name;};
    public Body getOrbits() {return orbits;}
}
```

**Note:** now the fields `idNum`, `name` and `orbits` are read-only outside the class. Methods that access internal data are called ***accessor methods*** sometime

# get / set methods

modification on the previous sample class with methods setting fields

```
class Body {
    private long idNum;
    private String name = "<unnamed>";
    private Body orbits = null;
    private static long nextID = 0;

    // constructors omitted for space problem. . .

    public long getID() {return idNum;}
    public String getName() {return name;};
    public Body getOrbits() {return orbits;}

    public void setName(String newName) {name = newName;}
    public void setOrbits(Body orbitsAround) {orbits =
orbitsAround;}
}
```

**Note:** now users can set the `name` and `orbits` fields. But `idNum` is still read-only

☞ Don't forget the `private` modifier on a data field when necessary! The default access modifier for fields is `package`

# Naming conventions

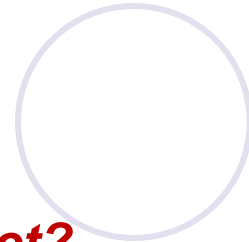
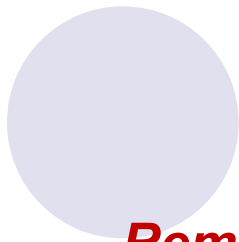
- **Package names:** start with lowercase letter
  - E.g. java.util, java.net, java.io . . .
- **Class names:** start with uppercase letter
  - E.g. File, Math . . .
  - avoid name conflicts with packages
  - avoid name conflicts with standard keywords in java system
- **Variable, field and method names:** start with lowercase letter
  - E.g. x, out, abs . . .
- **Constant names:** all uppercase letters
  - E.g. PI . . .
- **Multi-word names:** capitalize the first letter of each word after the first one
  - E.g. HelloWorldApp, getName . . .



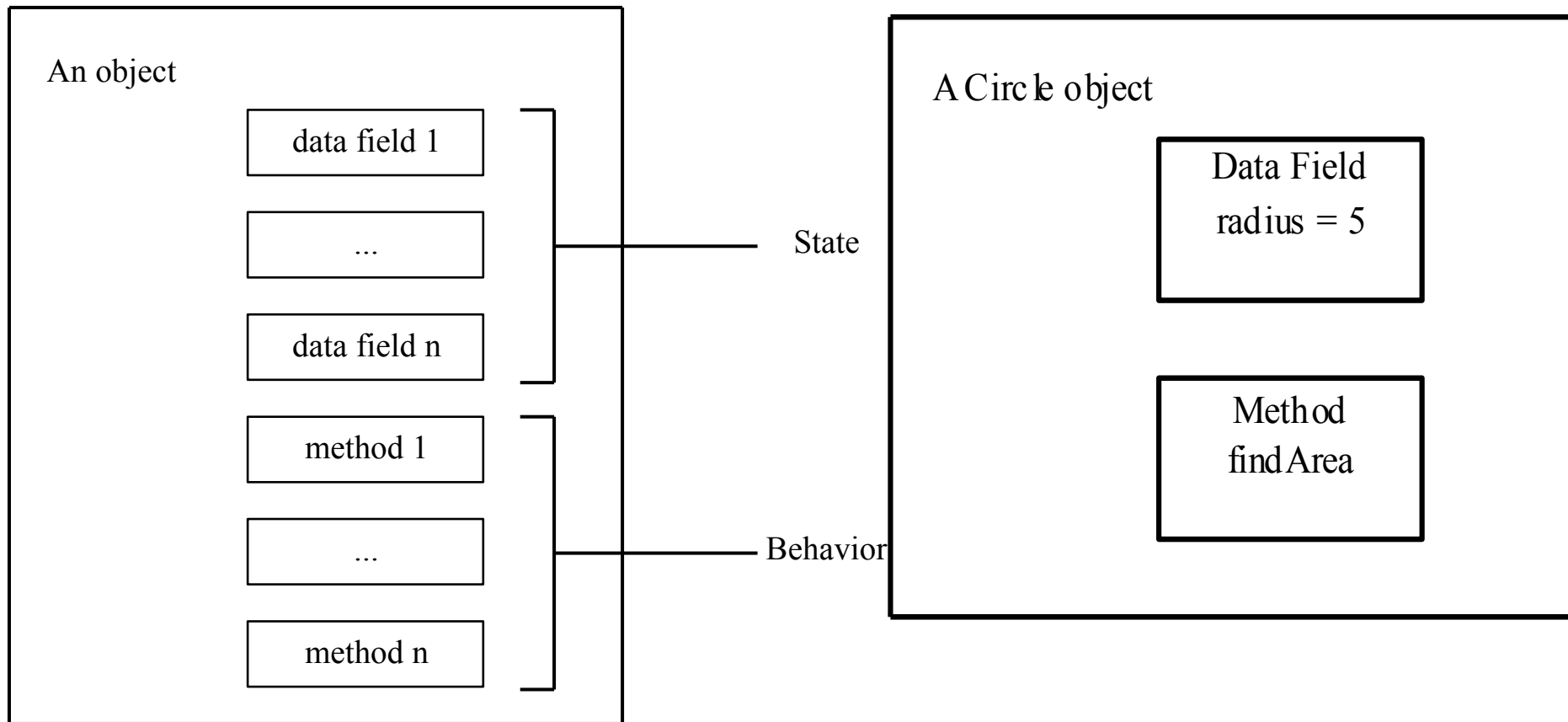
# Programming with Objects and Classes

- Declaring and Creating Objects
- Constructors
- Modifiers (`public`, `private` and `static`)
- Instance and Class Variables and Methods
- Scope of Variables
- Use the `this` Keyword
- Relationships among Classes
- The Java API and Core Java classes

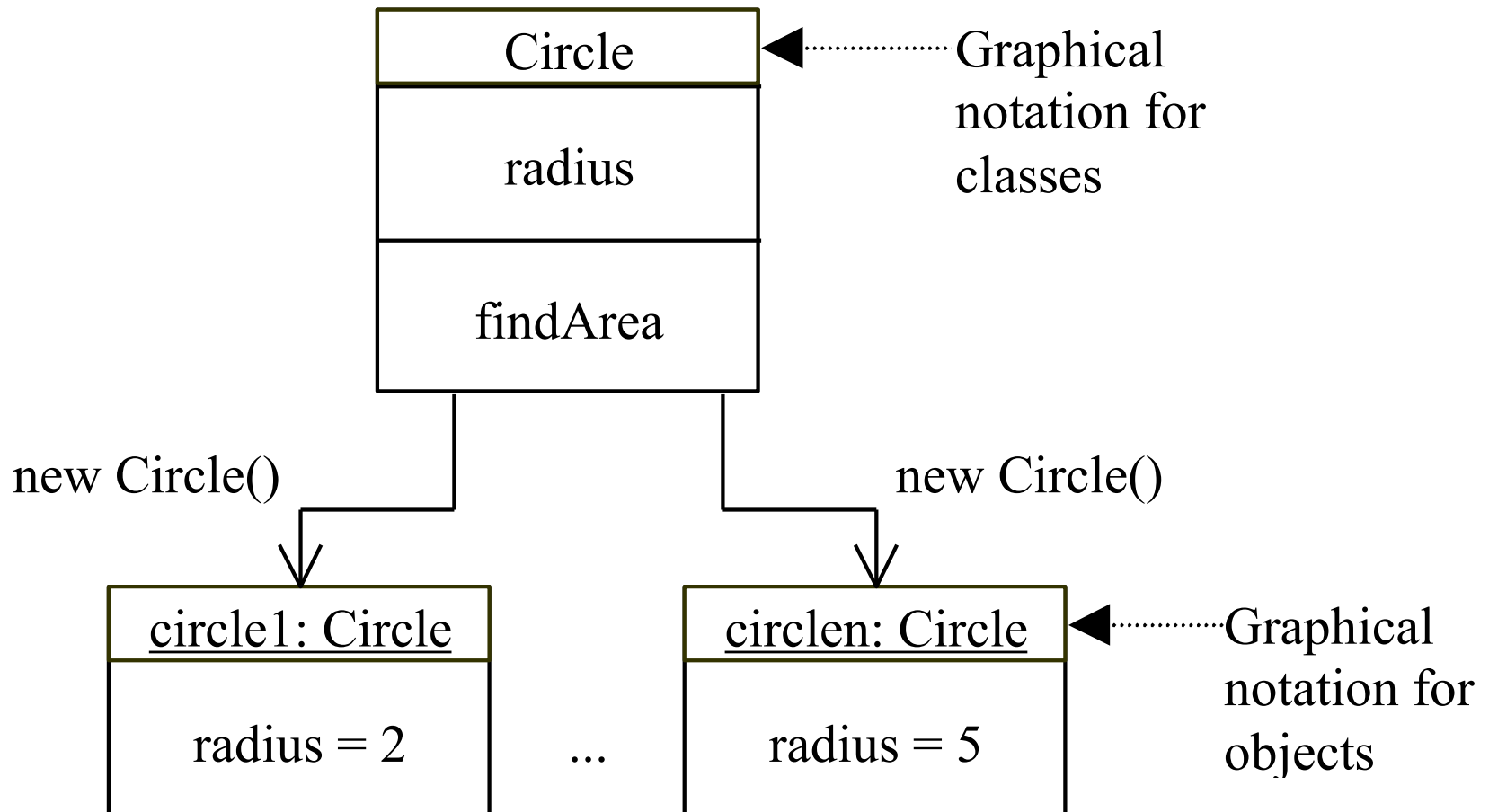




***Remember two main parts of any object?***



# Class and Objects



# Class Declaration

```
class Circle
{
    double radius = 1.0;

    double findArea()
    {
        return radius*radius*3.14159;
    }
}
```

# Declaring & Creating Objects

- **Declaration :**

```
ClassName objectName;  
Circle myCircle;
```

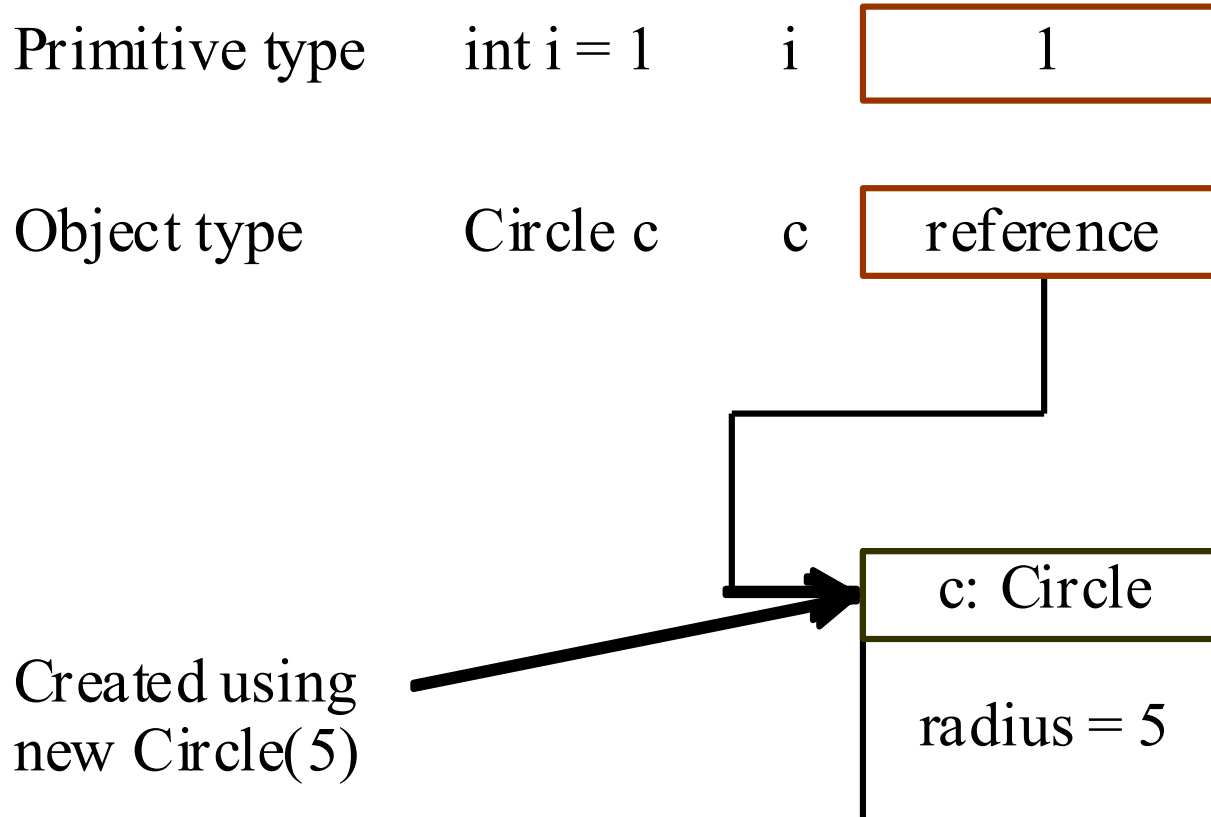
- **Creation:**

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

- **In one step:**

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

# Differences between variables of **primitive** and **reference** types



# Copying variables of **Primitive** and **Reference** types

Primitive type assignment  
 $i = j$

Object type assignment  
 $c1 = c2$

Before:

i 1

j 2

After:

i 2

j 2

Before:

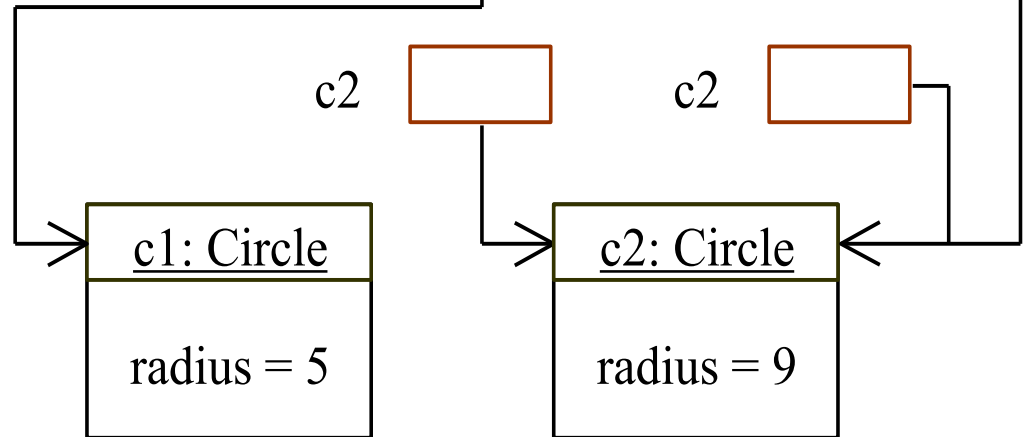
c1

c2

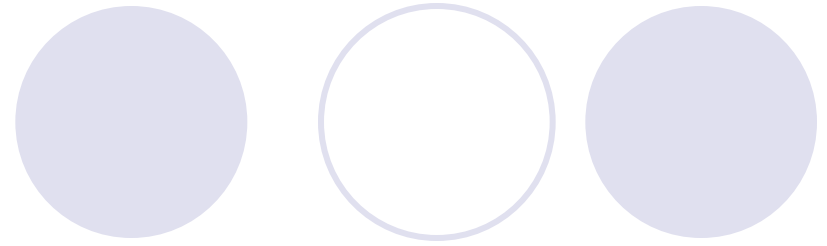
After:

c1

c2



# Constructors



```
Circle(double r)
```

```
{  
    radius = r;  
}
```

```
Circle()
```

```
{  
    radius = 1.0;  
}
```

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

```
Circle myCircle2 = new Circle(5.0) ;
```

# Remember Accessor Methods?

By default, the class, variable, or data 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 getter and setter accessor methods are used to read and modify private properties.





# Instance and Class Variables

- **Instance** variables belong to a specific **instance**.

**Instance** methods are invoked by an **instance** of the class.

- **Class** variables are shared by all instances of the class.

**Class** methods are not tied to a specific object.

To declare **class** variables, constants, and methods, use the **static** modifier.

# Class Variables and Methods

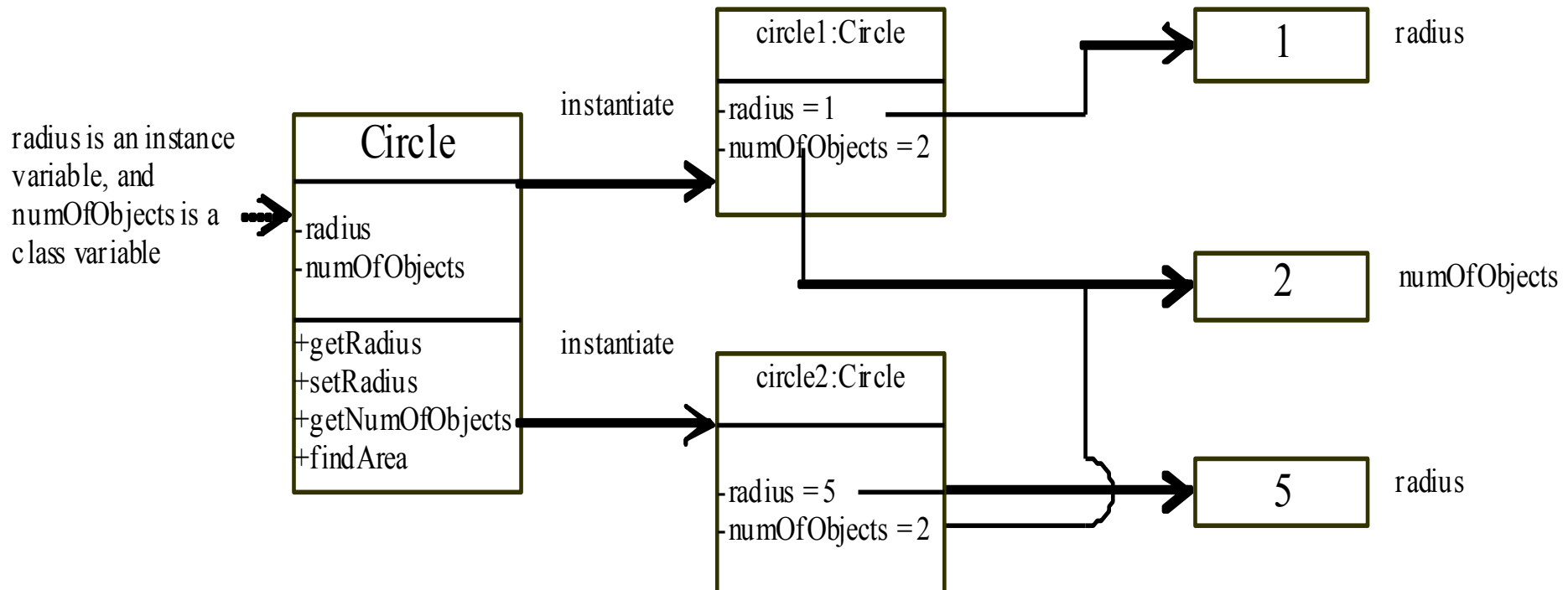
Notation:

+: public variables or methods

-: private variables or methods

underline: static variables or methods

Memory





# Relationships among Classes

- Association
- Aggregation
- Inheritance (Generalization)
- Realization
- Discussed later...

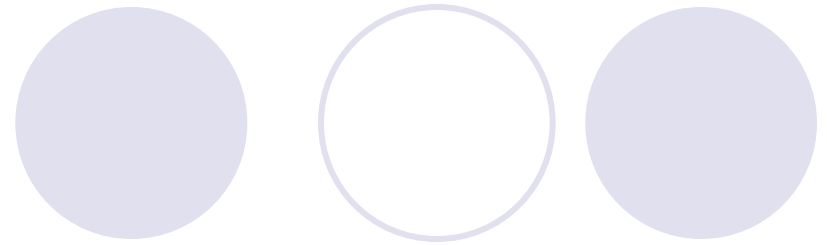


# Class Abstraction

Class **abstraction** means to **separate class implementation from the use of the class**. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



# Class Design



1. Identify classes for the system.
2. Describe attributes and methods
3. Establish relationships among classes.
4. Create classes.



# Java API and Core Java classes

- `java.lang`  
Contains core Java classes, such as numeric classes, strings, and objects. This package is implicitly imported to every Java program.
- `java.io`  
Contains classes for input and output streams and files.
- `java.util`  
Contains many utilities, such as date.
- `java.net`  
Classes for supporting network communications.