BIRZEIT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

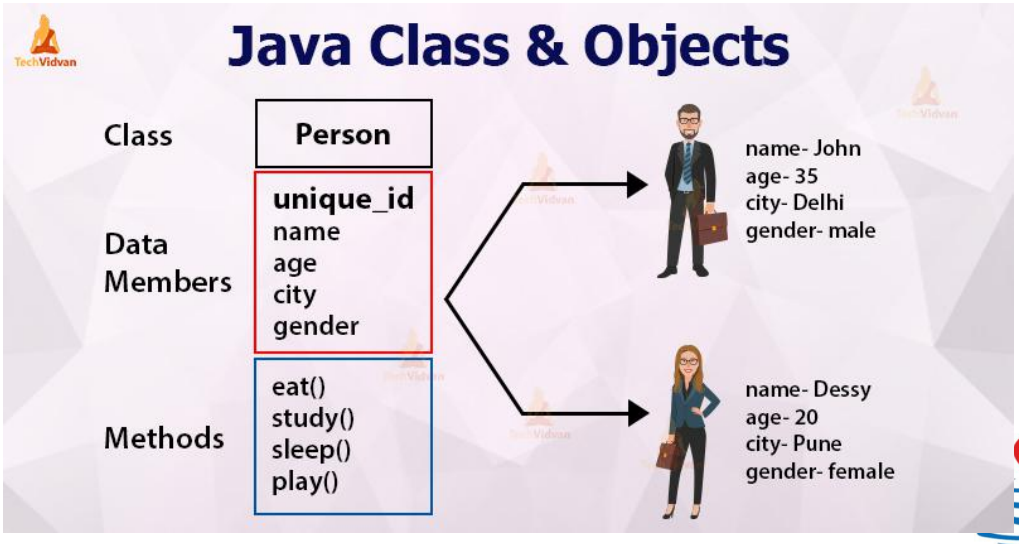
ADVANCED PROGRAMMING COMP231

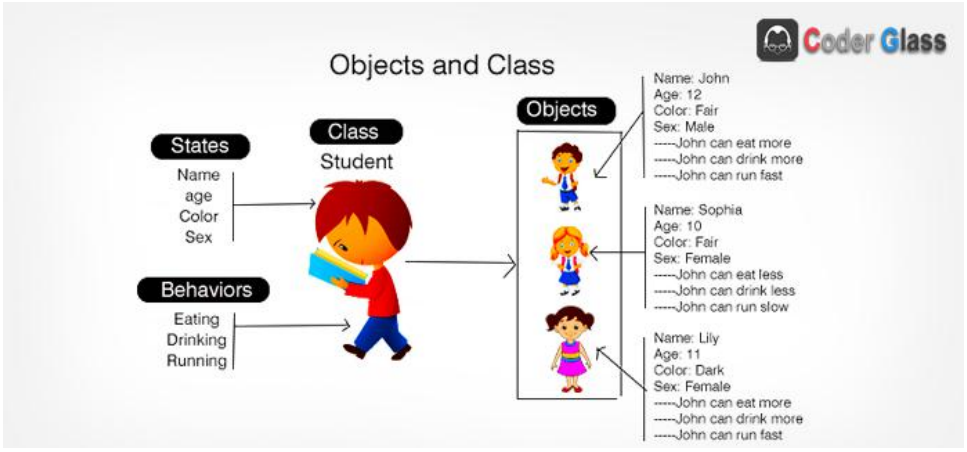
Instructor :Murad Njoum
Office : Masri322

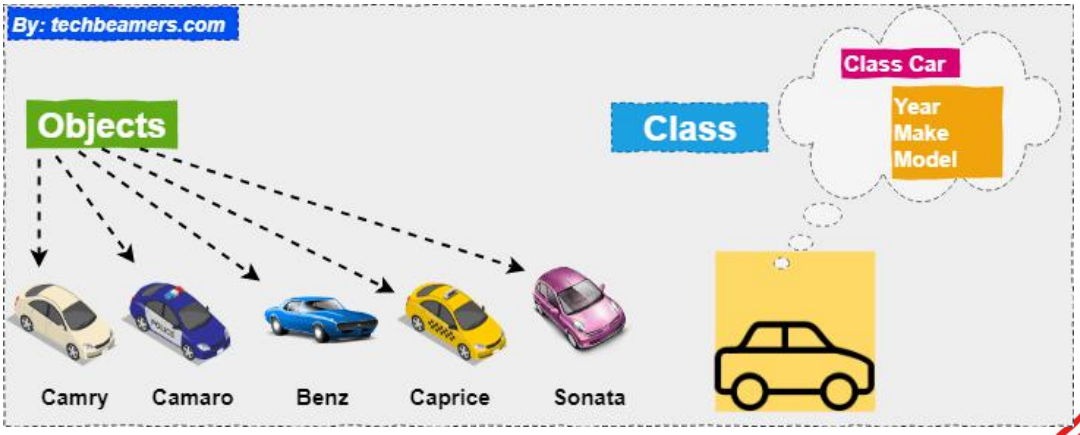
Chapter 9 Objects and Classes

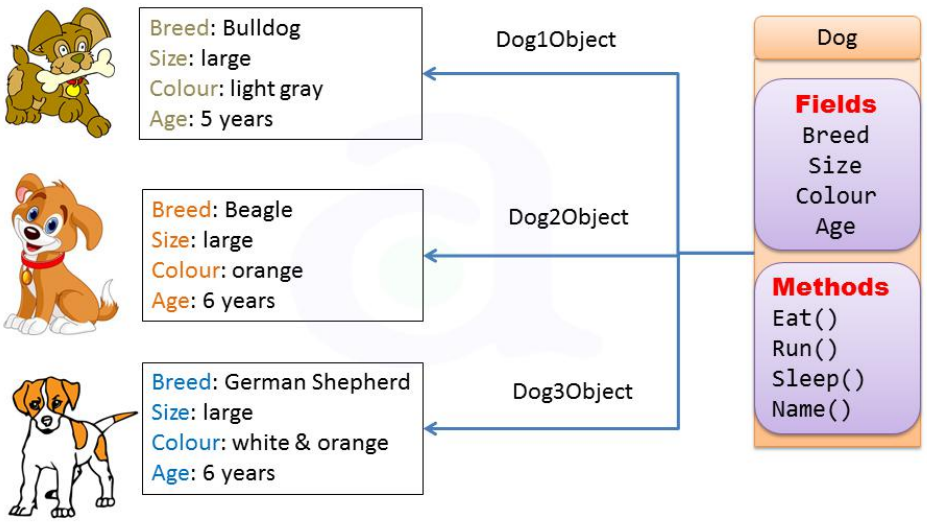
OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects. **An *object* represents an entity in the real world that can be distinctly identified**. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. **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.

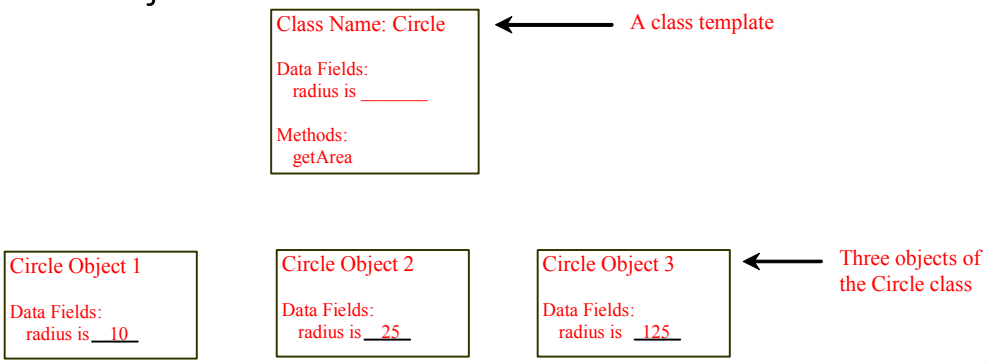

Java Class & Objects
Class
Person
Data Members
unique_id
name
age
city
gender
Methods
eat()
study()
sleep()
play()
name- John
age- 35
city- Delhi
gender- male
name- Dessy
age- 20
city- Pune
gender- female


Coder Glass
Objects and Class
States
Name
age
Color
Sex
Behaviors
Eating
Drinking
Running
Class
Student
Objects
Name: John
Age: 12
Color: Fair
Sex: Male
-----John can eat more
-----John can drink more
-----John can run fast
Name: Sophia
Age: 10
Color: Fair
Sex: Female
-----John can eat less
-----John can drink less
-----John can run slow
Name: Lily
Age: 11
Color: Dark
Sex: Female
-----John can eat more
-----John can drink more
-----John can run fast


By: techbeamers.com
Objects
Class
Class Car
Year
Make
Model
Camry
Camaro
Benz
Caprice
Sonata


Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years
Dog1Object
Breed: Beagle
Size: large
Colour: orange
Age: 6 years
Dog2Object
Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years
Dog3Object
Dog
Fields
Breed
Size
Colour
Age
Methods
Eat()
Run()
Sleep()
Name()

Objects

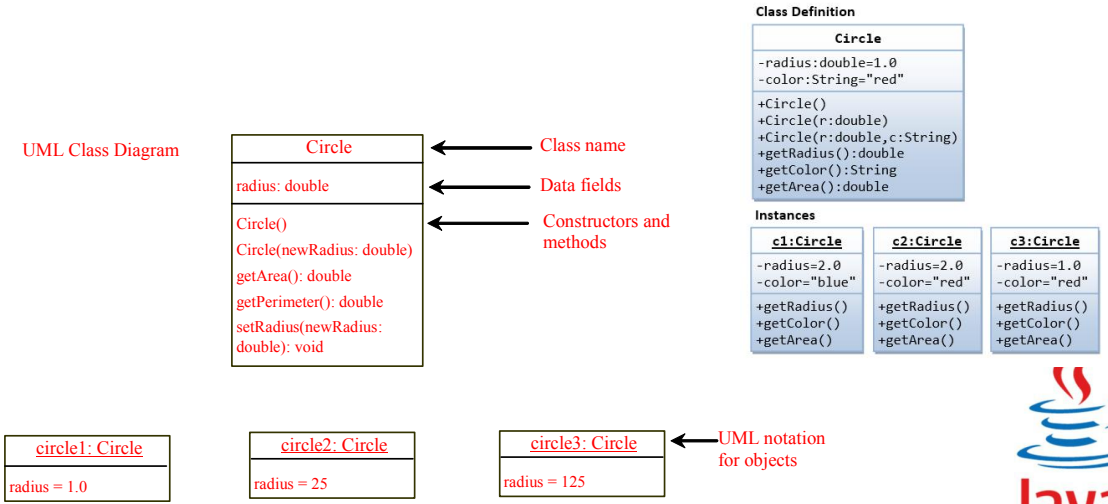


An object has **both a state and behavior**. The state defines the object, and the behavior defines what the object does.

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.

UML Class Diagram: Unified Modeling Language




```
class SimpleCircle {
  double radius;

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

  /** Construct a circle with a specified radius */
  SimpleCircle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

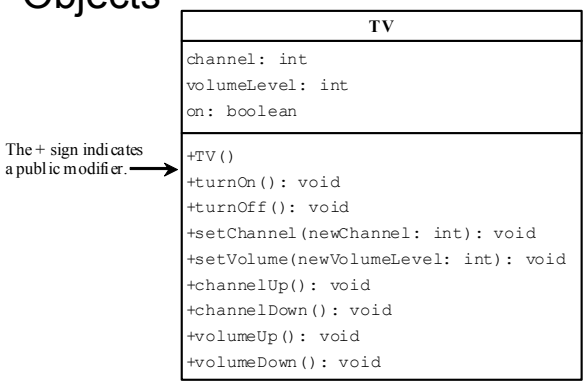
```
public class TestSimpleCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    SimpleCircle circle1 = new SimpleCircle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

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

    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }
}
```

Example: Defining Classes and Creating Objects



TV

TestTV Run

```
public class TV {
  int channel = 1; // Default channel is 1
  int volumeLevel = 1; // Default volume level is 1
  boolean on = false; // By default TV is off

  public TV() {
  }

  public void turnOn() {
    on = true;
  }

  public void turnOff() {
    on = false;
  }

  public void setChannel(int newChannel) {
    if (on && newChannel >= 1 && newChannel <= 120)
      channel = newChannel;
  }

  public void setVolume(int newVolumeLevel) {
    if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
      volumeLevel = newVolumeLevel;
  }

  public void channelUp() {
    if (on && channel < 120)
      channel++;
  }

  public void channelDown() {
    if (on && channel > 1)
      channel--;
  }

  public void volumeUp() {
    if (on && volumeLevel < 7)
      volumeLevel++;
  }

  public void volumeDown() {
    if (on && volumeLevel > 1)
      volumeLevel--;
  }
}
```


```
public class TestTV {
  public static void main(String[] args) {
    TV tv1 = new TV(); // Create a TV
    tv1.turnOn(); // Turn on tv1
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2 = new TV();
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp(); // Increase tv2 volume up 1 level

    System.out.println("tv1's channel is " + tv1.channel  + " and volume level is " + tv1.volumeLevel);
    System.out.println("tv2's channel is " + tv2.channel   + " and volume level is " + tv2.volumeLevel);
  }
}
```

Constructors

```
Circle() {
}
```

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

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

Constructors, cont.

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;
```

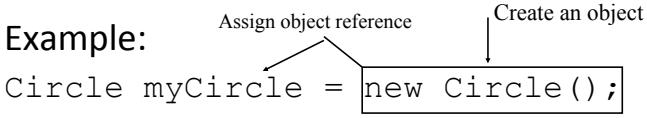
Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

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

Example:



Accessing Object's Members

❑Referencing the object's data:

```
objectRefVar.data
```

e.g., `myCircle.radius`

❑Invoking the object's method:

```
objectRefVar.methodName(arguments)
```

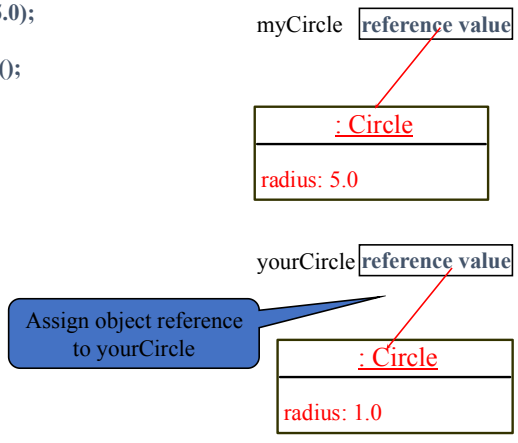
e.g., `myCircle.getArea()`

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

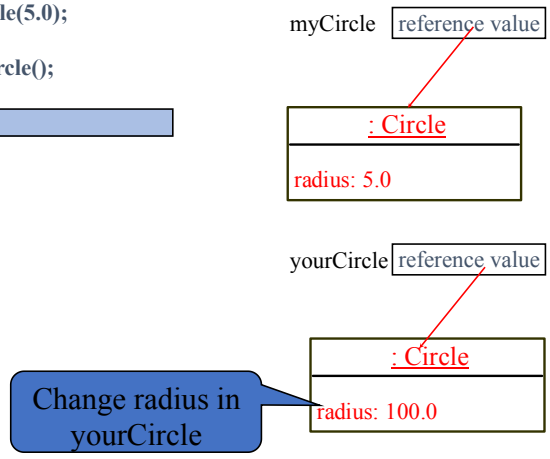


Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Caution

Recall that you use

Math.methodName(arguments) (e.g., Math.pow(3, 2.5))

to invoke a method in the Math class

Can you invoke getArea() using SimpleCircle.getArea()? The answer is no. All the methods used before this chapter **are static methods**, which are defined using the static keyword. However, **getArea() is non-static**. It must be invoked from an object using

objectRefVar.methodName(arguments) (e.g., 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
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // c has default value '\u0000'
}
```

The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

Default Value for a Data Field

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```
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);
  }
}
```

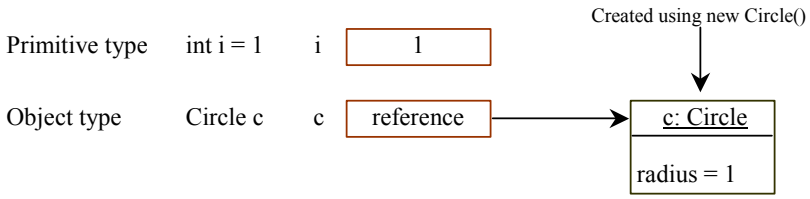
Example

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);
  }
}
```

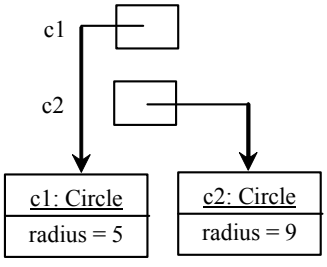
Compile error: variable not initialized

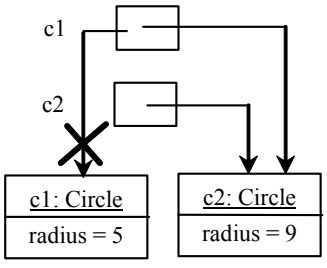
Differences between Variables of Primitive Data Types and Object Types



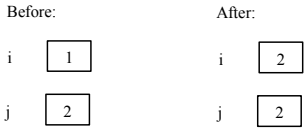
Copying Variables of Primitive Data Types and Object Types

Object type assignment c1 = c2

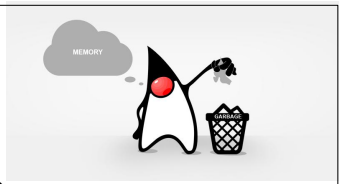




Primitive type assignment i = j



Garbage Collection



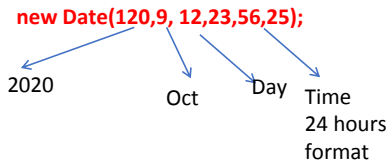
As shown in the previous figure, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. **Garbage is automatically collected by JVM.**

Garbage Collection,

TIP: 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.**



The + sign indicates public modifer

java.util.Date	
+Date()	Constructs a Date object for the current time.
+Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
+toString(): String	Returns a string representing the date and time.
+getTime(): long	Returns the number of milliseconds since January 1, 1970, GMT.
+setTime(elapseTime: long): void	Sets a new elapse time in the object.

The Date Class Example

For example, the following code

```
java.util.Date date = new java.util.Date();
  //current date
System.out.println(date.toString());
```

displays a string like `Mon Oct 12 19:10:18 IDT 2020`.

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 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 will generate identical sequences of numbers. For example, the following code creates two Random objects with the same **seed 3**.

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

Static Variables, Constants, and Methods

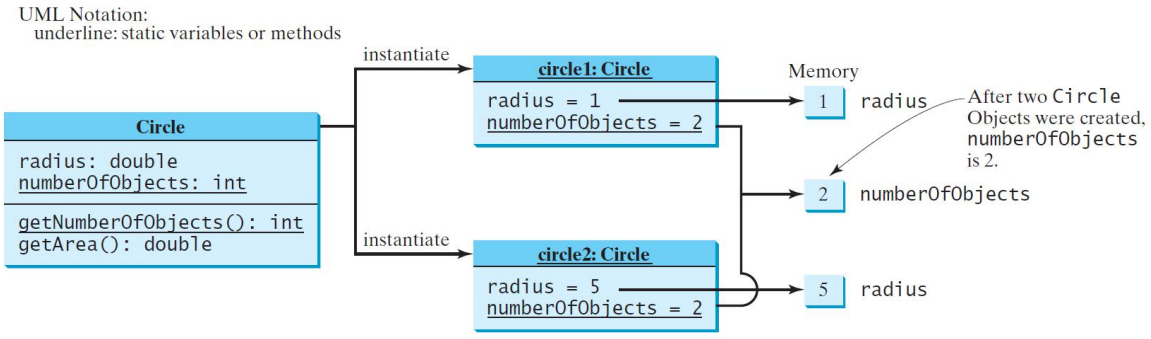
Static variables are shared by all the instances of the class.

Static methods are not tied to a specific instance (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, cont.



Example of Using Instance and Class Variables and Method

Objective: Demonstrate the roles of instance and class variables and their uses. This example adds a class variable numberOfObjects to track the number of Circle objects created.

CircleWithStaticMembers

TestCircleWithStaticMembers

Run


```
public class CircleWithStaticMembers {
  /** The radius of the circle */
  double radius;

  /** The number of the objects created */
  static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  CircleWithStaticMembers() {
    radius = 1.0;
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  CircleWithStaticMembers(double newRadius)
  {
    radius = newRadius;
    numberOfObjects++;
  }

  /** Return numberOfObjects */
  static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Static Variable

1. It is a **variable** which belongs to the **class** and **not to the instance** (object).
2. Static variables are initialized **only once**, at the **start** of the execution.
Static variables will be **initialized first, before** the initialization of any instance variables.
3. A **single copy** to be shared by **all instances of the class**.
4. A static variable can be **accessed directly** by the **class name** and doesn't **need any instance of class (object)**.

Syntax : < class - name>.<static - variable - name>

Static Method

1. It is a method which **belongs to the class and not to the instance** (object).
2. A static method **can access only static data**. It can not access **non-static** data (instance variables).
3. A static method **can call only other static** methods and **can not call a non-static method from method inside. (main and other methods inside class)**
4. A static method can be accessed directly by the class name and doesn't need any create an instance (object) to access it.
Syntax : < class - name>.<static - method - name>(..)
5. **A static method cannot refer to "this" or "super" keywords in anyway.**

Note: main method is static, since it must be accessible for an application to run, **before any instantiation takes place**.

```
public class Checkstatic {
        public static void main(String[] args) {

        Check c1 = new Check();
        System.out.println(c1.getX());
        System.out.println(c1.x); // warining: static field should be accessed in static way

        Check c2 = new Check();
        System.out.println(c2.getX());
        System.out.println(c2.x); // warining: static field should be accessed in static way
        }
}

class Check {
        static int x = 0;
        int y;

        Check() {
                x++;  }

        Check(int xvalue) {
                y = xvalue;
                x++;
        }

        public int getX() {
                return x;
        }
}
```

The static field Check.x should be accessed in a static way

1
1
2
2


```
public class Checkstatic {

public static void main(String[] args) {


      System.out.println(Check.x);

       Check.setX(5);

     System.out.println(Check.getX());
    }
}

class Check{
static int x=0;
 int y;
Check(){

  }

public static void setX(int xvalue){
      x=xvalue;
      }
public static int  getX(){
     return x;
    }
}
```

We accessed in a static way
Since no instance object createted

0
5

```
public class Checkstatic {

public static void main(String[] args) {


    System.out.println(Check.x);

     Check.setX(5); //error: cannot make static reference to non-static

   System.out.println(Check.getX());//error: cannot make static reference to non-static
  }
}

class Check{
static int x=0;
 int y;
Check(){

  }

public  void setX(int xvalue){
    x=xvalue;
    }
public  int  getX(){
    return x;
   }
}
```

Visibility Modifiers and Accessor/Mutator Methods

"A Mutator method is commonly known as a set method or simply a setter"

"It shows us the principle of encapsulation"

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.(variables)


```
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();
  }
}
```

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.

Most Restrictive ⟵ ⟶ Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Same rules apply for inner classes too, they are also treated as outer class properties

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

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

NOTE

An object cannot access its private members, as shown in (b).
It is OK, however, if the object is **declared in its own class**, as shown in (a).

```
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;
  }
}
```

(a) This is okay because object `c` is used inside the class `C`.

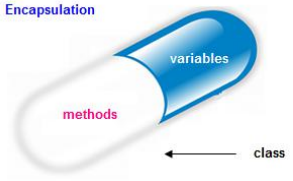
```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

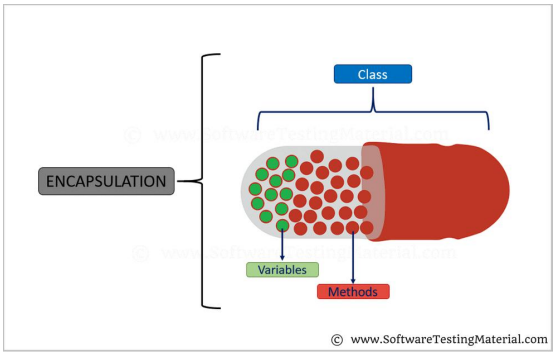
(b) This is wrong because `x` and `convert` are private in class `C`.

Why Data Fields Should Be private?

To protect data.

To make code easy to maintain.





Example of Data Field Encapsulation

The - sign indicates private modifier

Circle	
-radius: double	The radius of this circle (default: 1.0).
-numberOfObjects: int	The number of circle objects created.
+Circle()	Constructs a default circle object.
+Circle(radius: double)	Constructs a circle object with the specified radius.
+getRadius(): double	Returns the radius of this circle.
+setRadius(radius: double): void	Sets a new radius for this circle.
+getNumberOfObjects(): int	Returns the number of circle objects created.
+getArea(): double	Returns the area of this circle.

CircleWithPrivateDataFields

TestCircleWithPrivateDataFields Run

Passing Objects to Methods

```
public class TestPassObject {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle object with radius 1
    CircleWithPrivateDataFields myCircle = new CircleWithPrivateDataFields(1);
    // Print areas for radius 1, 2, 3, 4, and 5.
    int n = 5;
    printAreas(myCircle, n);

    // See myCircle.radius and times
    System.out.println('\n' + 'Radius is ' + myCircle.getRadius());
    System.out.println('n is ' + n);
  }
/** Print a table of areas for radius */
  public static void printAreas(CircleWithPrivateDataFields c, int times) {
    System.out.println('Radius \t\tArea');
    while (times != 1) {
      System.out.println(c.getRadius() + '\t\t' + c.getArea());
      c.setRadius(c.getRadius() + 1);
      times--;
    }
  }
}
```

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)

```
Radius        Area
1.0           3.141592653589793
2.0           12.566370614359172
3.0           28.274333882308138
4.0           50.26548245743669
5.0           78.53981633974483
Radius is 6.0n is 5
```

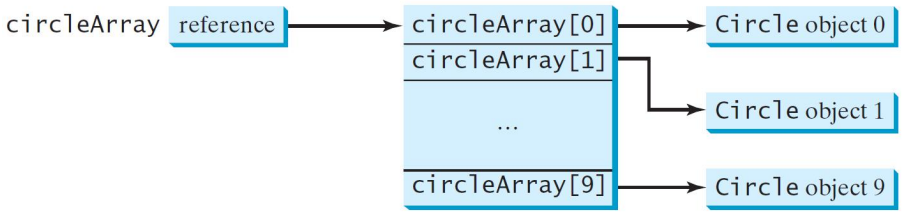
Array of Objects

```
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 as shown in the next figure. circleArray references to the entire array. circleArray[1] references to a Circle object.

Array of Objects, cont.

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



Immutable(Cannot change) 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 class in Listing 8.10, **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 (no setter)** is not necessarily **immutable.** For example, the following class Student has all private data fields and **no mutators**, but it is **mutable.**

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;
  }
}
```

```
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;
  }
}
```

```
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!
  }
}
```

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.

Immutable Class غير قابل للتغير

- Immutable class means that once an object is created, **we cannot change its content.**
- The class must be declared as **final** (So that child classes can't be created)
- Data members in the **class must be declared as final** (So that we can't change the value of it after object creation)
- A parameterized constructor
- **Getter method for all the variables in it**
- **No setters**(To not have the option to change the value of the instance variable)

Example

```
// An immutable class
public final class Student
{ final String name;
    final int regNo;

        public Student(String name, int regNo)
        {       this.name = name;
                this.regNo = regNo;
        }
        public String getName()
        {  return name; }
        public int getRegNo()
        { return regNo;      }
}
```

```
// Driver class
class Test
{
      public static void main(String args[])
      {
            Student s = new Student("ABC", 101);
            System.out.println(s.getName());
            System.out.println(s.getRegNo());

            // Uncommenting below line causes error
            // s.regNo = 102;
      }
}
```

- 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**.

Remember : . A static method cannot refer to "this" or "super" keywords in anyway.

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