Note #1 – Object Oriented Programming

- **Object-Oriented Programming**

Object-oriented programming (OOP) enables you to develop large-scale software and Graphical User Interfaces (GUIs) effectively. It is essentially a technology for developing reusable software. Having learned Java programming language in the previous courses, you are able to solve many computer solvable problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing GUI and large-scale software systems.

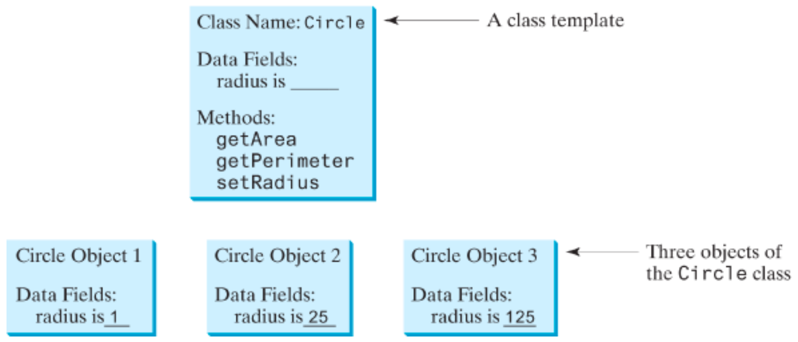
- **Class and object**

A **class** defines the properties and behaviors for objects. 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 behavior.

The **state** of an object (also known as its properties or attributes) is represented by data fields with their current values. A circle object, for example, has a data field radius, which is the property that characterizes a circle. A rectangle object, for example, has the data fields width and height, which are the properties that characterize a rectangle.

The **behavior** of an object (also known as its actions) is defined by methods. To invoke a method on an object is to ask the object to perform an action, which is to manipulate the data fields. For example, you may define methods named **getArea()** and **getPerimeter()** for circle objects. A circle object may invoke **getArea()** to return its area and **getPerimeter()** to return its perimeter. You may also define the **setRadius(radius)** method. A circle object can invoke this method to change its radius.

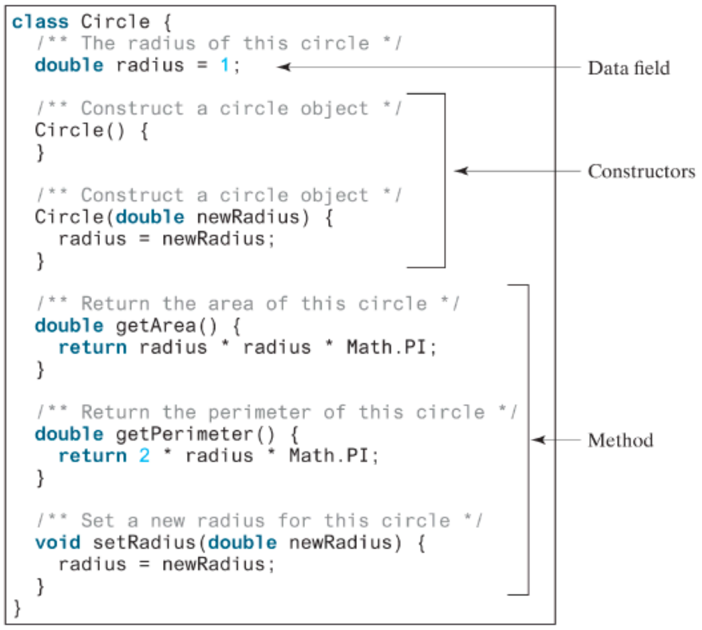
Objects of the same type are defined using a common class. **A class is a template**, blueprint, or contract that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as **instantiation**. The terms object and instance are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe.



- **Class and Object Implementation with Java**

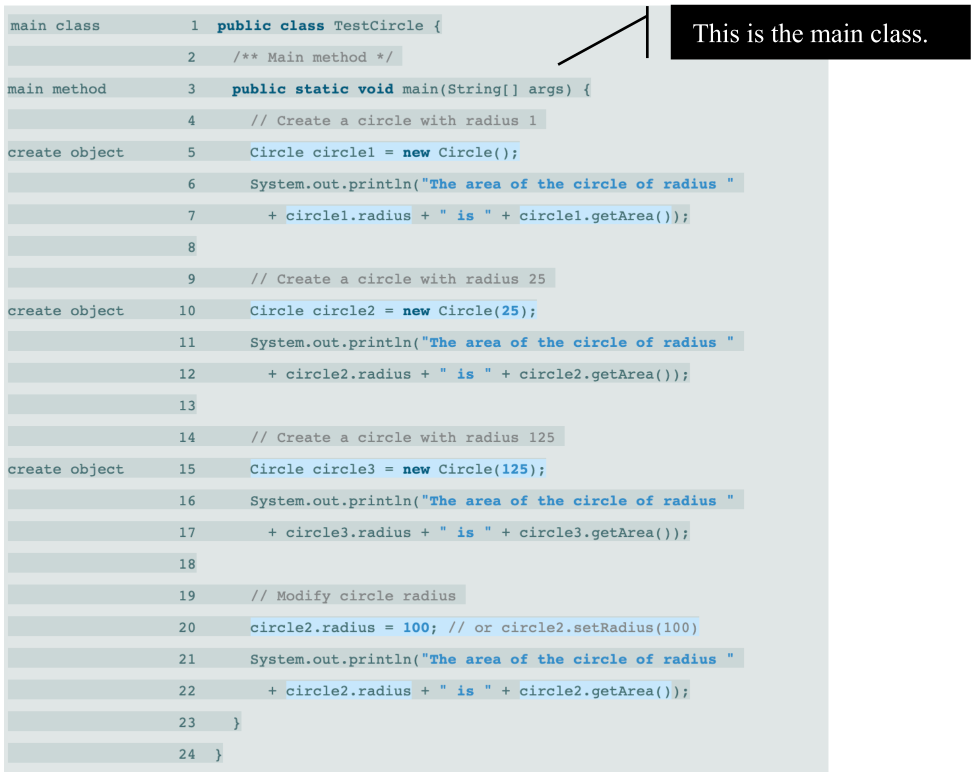
Unlike C++, Java is a pure object-oriented programming language. A Java class uses variables to define data fields and methods to define actions. In addition, a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Because the data fields are mostly "private" and only visible within the class.

For example, the Circle class below is a template for creating objects with different "states". Note that, the Circle class doesn't have a "main" method and cannot be run by itself.

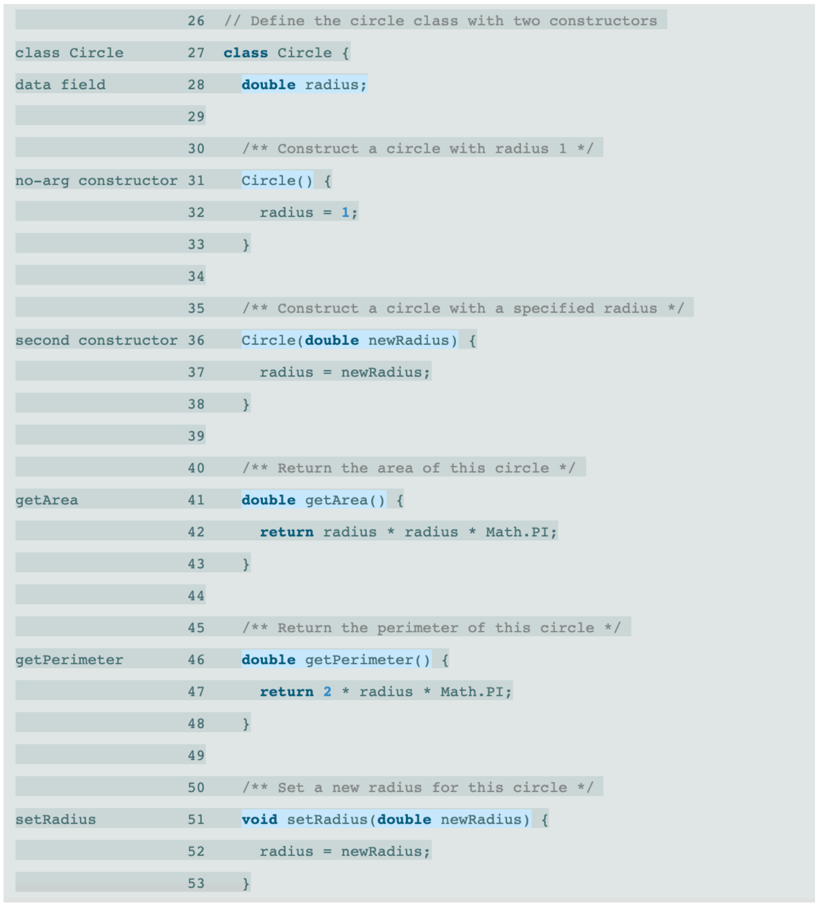


```java
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** 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 * 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;
  }
}
```

- **An example of defining classes and creating Objects with Java**



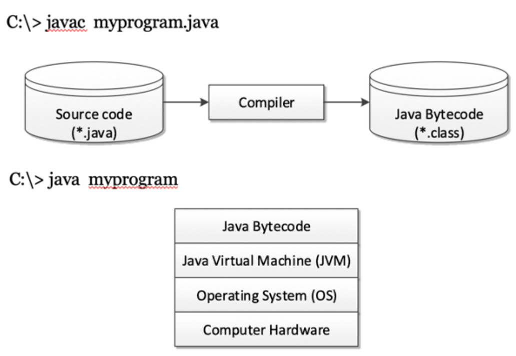
```java
public class TestCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    Circle circle1 = new Circle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

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

    // Create a circle with radius 125
    Circle circle3 = new Circle(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());
  }
}
```



```
// Define the circle class with two constructors
class Circle {
  double radius;

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

  /** Construct a circle with a specified radius */
  Circle(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;
  }
```

The program contains two classes. The first of these, TestCircle, is the main class. Its sole purpose is to test the second class, Circle. Such a program that uses the class is often referred to as **a client of the class**. When you run the program, the Java runtime system invokes the main method in the main class.

You can put the two classes into one file, but only one class in the file can be a public class. Furthermore, **the public class must have the same name as the file name**. Therefore, the file name of the main class would be **TestCircle.java** in this case, since TestCircle is public. Each class in the source code is compiled into a .class file. When you compile TestCircle.java, two class files TestCircle.class and Circle.class are generated. Note that, Java uses a combination of a compiler an interpreter. Java programs compiled into bytecode, which is interpreted and run by JVM (Java Virtual Machine.)

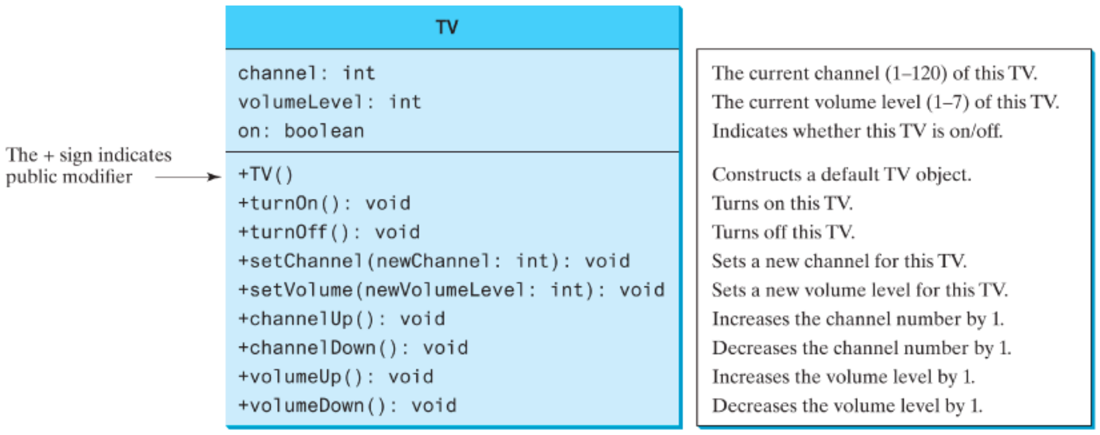


```
C:\> javac myprogram.java
```

```
C:\> java myprogram
```

There are many ways to write Java programs. For instance, you can combine the two classes in the preceding example into one. Since the combined class has a **main** method, it can be executed by the Java interpreter. This demonstrates that you can test a class by simply adding a **main** method in the same class. In this case, we call the main method as "testbed main".

As another example, consider television sets. Each TV is an object with states (current channel, current volume level, and power on or off) and behaviors (change channels, adjust volume, and turn on/off). You can use a class to model TV sets.



TV
channel: int
volumeLevel: int
on: boolean
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
The current channel (1–120) of this TV.
The current volume level (1–7) of this TV.
Indicates whether this TV is on/off.
Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.
The + sign indicates public modifier

- **Constructors**

A constructor is invoked to create (instantiate) an object using the "new" operator. Constructors are a special kind of method. They have three peculiarities:

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

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the "new" operator, as follows: **new ClassName(arguments);**

For example, **new Circle()** creates an object of the Circle class using the first constructor defined in the Circle class, and **new Circle(25)** creates an object using the second constructor defined in the Circle class. A class normally provides a constructor without arguments (e.g., Circle()). Such a constructor is referred to as a no-arg or no-argument constructor. A class may be defined without constructors. In this case, a public 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.

- **Accessing Objects**

An object's data and methods can be accessed through the dot (.) operator via the object's reference variable. Newly created objects are allocated in the memory. They can be **accessed via reference variables**. Objects are accessed via the object's reference variables, which contain references to the objects. A class is essentially a programmer-defined type. A class is a reference type, which means that a variable of the class type can reference an instance of the class. You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

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

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that **myCircle** is a Circle object rather than use the long-winded description that **myCircle** is a variable that contains a reference to a Circle object.

In OOP terminology, an object's member refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the dot operator (.), also known as the object member access operator.

In the Circle class, the data field **radius** is referred to as an ***instance variable*** because it is dependent on a specific instance. For the same reason, the method **getArea** is referred to as an ***instance method*** because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

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 **Circle.getArea()?** The answer is NO. All the methods in the Math class are static methods, which are defined using the static keyword. However, **getArea()** is an instance method, and thus non-static. It must be invoked from an object using objectRefVar.methodName(arguments) (e.g., myCircle.getArea()).

Usually you create an object and assign it to a variable, then later you can use the variable to reference the object. Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a Circle object. The latter creates a Circle object and invokes its getArea method to return its area. An object created in this way is known as an anonymous object. The data fields can be of reference types. For example, the following Student class contains a data field name of the String type. String is a predefined Java class.

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

If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**. **null** is a literal just like **true** and **false**. While **true** and **false** are Boolean literals, **null** is a literal for a reference type. 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. The following code displays the default values of the data fields name, age, isScienceMajor, and gender for a Student object:

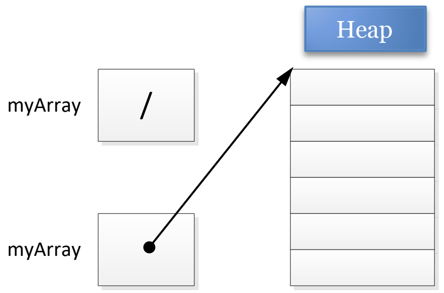
```
class TestStudent {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender?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

The following code has a compile error.

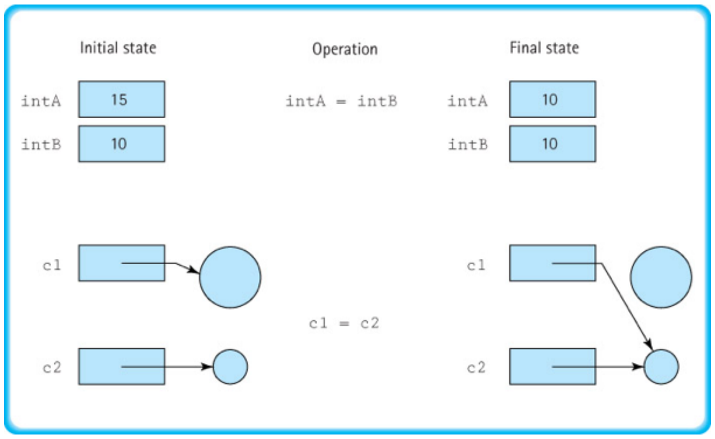
```
class TestLocalVariables {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

NullPointerException is a common runtime error. It occurs when you invoke a method on a reference variable with a **null** value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located.



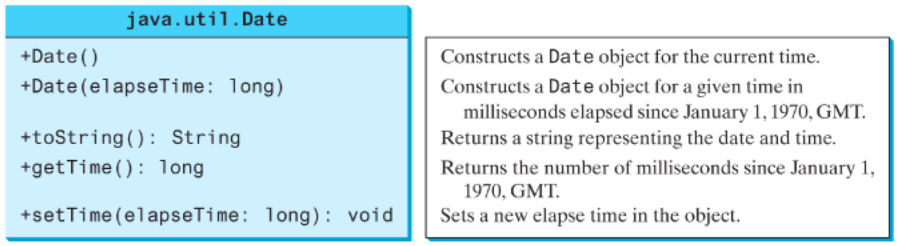
When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable.



After the assignment statement **c1 = c2**, **c1** points to the same object referenced by **c2**. The object previously referenced by **c1** is no longer accessible and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

- **Using Java library classes**

For example, the **Date class**

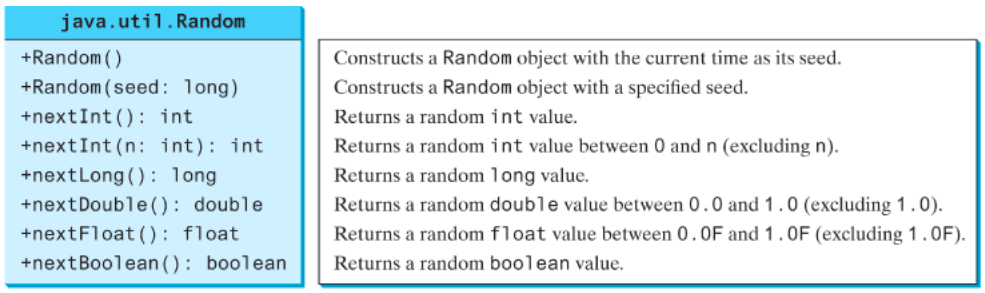


java.util.Date	
+Date()	Constructs a Date object for the current time.
+Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
+toString(): String	Returns a string representing the date and time.
+getTime(): long	Returns the number of milliseconds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

You can use the no-arg constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time in milliseconds since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string. For example, the following code.

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " + date.getTime() + " milliseconds");
System.out.println(date.toString());
```

Another useful class, the **Random class**

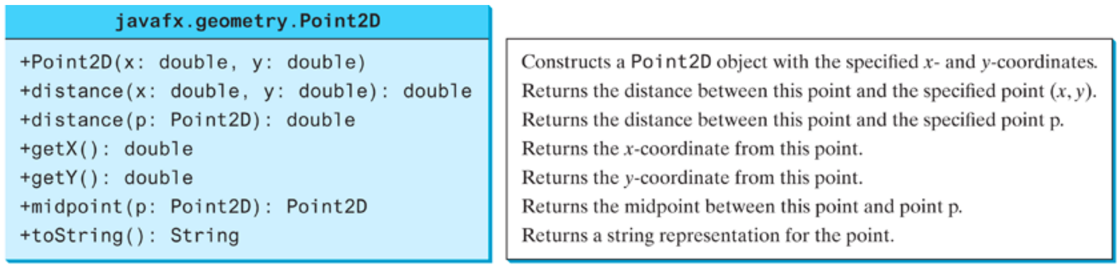


java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long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

When you create a Random object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a Random object using the current elapsed time as its seed. If two Random objects have the same seed, they will generate identical sequences of numbers. The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, often you need to reproduce the test cases from a fixed sequence of random numbers.

You can generate random numbers using the java.security.SecureRandom class rather than the Random class. The random numbers generated from the Random are deterministic and they can be predicated by hackers. The random numbers generated from the SecureRandom class are nondeterministic and are secure.

The **Point2D class**



javafx.geometry.Point2D	
+Point2D(x: double, y: double)	Constructs a Point2D object with the specified *x*- and *y*-coordinates.
+distance(x: double, y: double): double	Returns the distance between this point and the specified point (*x*, *y*).
+distance(p: Point2D): double	Returns the distance between this point and the specified point p.
+getX(): double	Returns the *x*-coordinate from this point.
+getY(): double	Returns the *y*-coordinate from this point.
+midpoint(p: Point2D): Point2D	Returns the midpoint between this point and point p.
+toString(): String	Returns a string representation for the point.

For a list of Java API library, please visit:
https://docs.oracle.com/en/java/javase/13/docs/api/index.html

- **Static variables, constants and methods**

A static variable is shared by all objects of the class. A static method cannot access instance members (i.e., instance data fields and methods) of the class.

The data field **radius** in the circle class is known as an instance variable. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. If you want all the instances of a class to share data, use static variables, also known as **class variables**. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. Static methods can be called without creating an instance of the class.

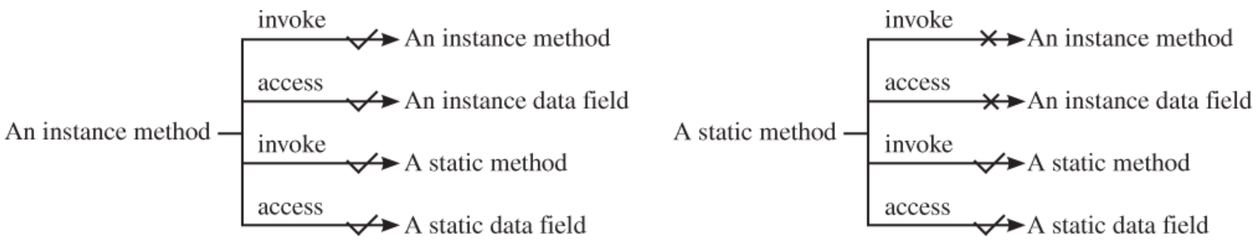
To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration.

Constants in a class are shared by all objects of the class. Thus, constants should be declared as **final static.** For example, the constant PI in the Math class is defined as follows:

final static double PI = 3.14159;

The main method is static, too. Static variables and methods can be accessed without creating objects. Use **ClassName.methodName(arguments)** to invoke a static method and **ClassName.staticVariable** to access a static variable. This improves readability because this makes static methods and data easy to spot.

An instance method can invoke an instance or static method, and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object. The relationship between static and instance members is summarized in the following diagram:



How do you decide whether a variable or a method should be instance or static? A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, radius is an instance variable of the Circle class. Since the getArea method is dependent on a specific circle, it is an instance method. None of the methods in the Math class, such as random, pow, sin, and cos, is dependent on a specific instance. Therefore, these methods are static methods. The main method is static and can be invoked directly from a class. It is a common design error to define an instance method that should have been defined as static. For example, the method **factorial(int n)** should be defined as static, because it is independent of any specific instance.

- **Visibility Modifiers**

Visibility modifiers can be used to specify the visibility of a class and its members. You can use the public visibility modifier for classes, methods, and data fields to denote they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as package-private or package-access

Packages can be used to organize classes. To do so, you need to add the following line as the first non-comment and nonblank statement in the program:

If a class is defined without the package statement, it is said to be placed in the *default package*. Java recommends that you place classes into packages rather than using a default package. In addition to the **public** and default visibility modifiers, Java provides the **private** and **protected** modifiers for class members. This section introduces the **private** modifier. The **private** modifier makes methods and data fields accessible only from within its own class. if a class is not defined as public, it can be accessed only within the same package.

	Within the Class	Within Subclasses in the Same Package	Within Subclasses in Other Packages	Everywhere
public	✓	✓	✓	✓
protected	✓	✓	✓	
package	✓	✓		
private	✓			

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class. The private modifier applies only to the members of a class. The public modifier can apply to a class or members of a class. Using the modifiers public and private on local variables would cause a compile error.

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a private constructor. For example, there is no reason to create an instance from the Math class, because all of its data fields and methods are static. To prevent the user from creating objects from the Math class, the constructor in java.lang.Math is defined as follows:

```
private Math() {
}
```

- **Data Encapsulation**

Making data fields private protects data and makes the class easy to maintain.

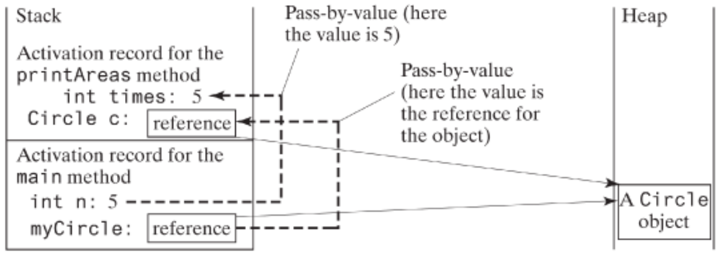
1. Data may be tampered with. For example, numberOfObjects is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., Circle.numberOfObjects = 10). The class becomes difficult to maintain and vulnerable to bugs.
2. Suppose that you want to modify the Circle class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the Circle class but also the programs that use it because the clients may have modified the radius directly (e.g., c1.radius = –5).

To prevent direct modifications of data fields, you should declare the data fields private, using the private modifier. This is known as data field encapsulation.

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a getter method to return its value. To enable a private data field to be updated, provide a setter method to set a new value. A getter method is also referred to as an accessor and a setter to a mutator. A getter method has the following signature:

- **Passing Objects to Methods**

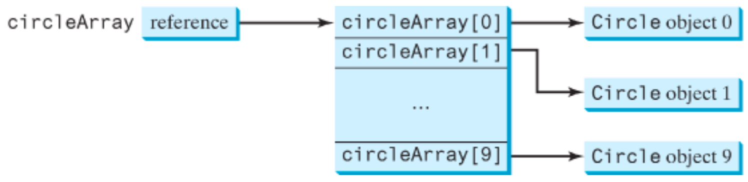
Passing an object to a method is to pass the reference of the object. You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the **myCircle** object as an argument to the **printCircle** method:


Stack
Activation record for the printAreas method
int times: 5
Circle c: reference
Activation record for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of n (5) is passed to times. Inside the printAreas method, the content of times is changed; this does not affect the content of n. When passing an argument of a reference type, the reference of the object is passed. In this case, c contains a reference for the object that is also referenced via myCircle. Therefore, changing the properties of the object through c inside the printAreas method has the same effect as doing so outside the method through the variable myCircle. Pass-by-value on references can be best described semantically as pass-by-sharing; that is, the object referenced in the method is the same as the object being passed.

- **Array of Objects**

An array of objects is actually an array of reference variables. Thus, invoking circleArray[1] .getArea() involves two levels of referencing, as shown below. circleArray references the entire array, and circleArray[1] references a Circle object.

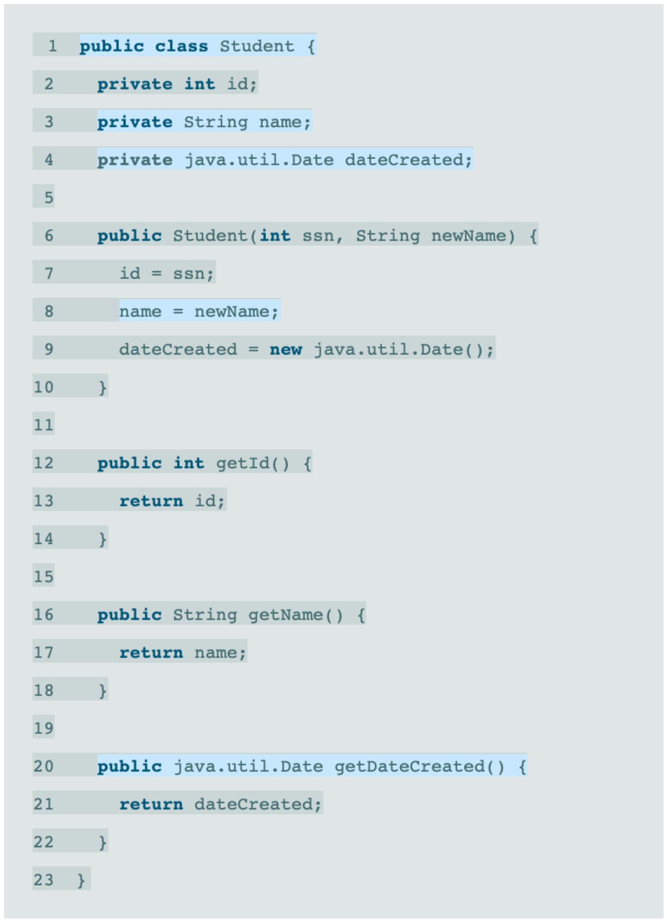


When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

- **Immutable Objects and Classes**

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed. Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as immutable object and its class as immutable class. The String class, for example, is immutable. If you deleted the setter method in the Circle class, the class would be immutable because radius is private and cannot be changed without a setter method.

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable. For example, the following **Student** class has all private data fields and no setter methods, but it is not an immutable class:



```java
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;

  public Student(int ssn, String newName) {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public java.util.Date getDateCreated() {
    return dateCreated;
  }
}
```

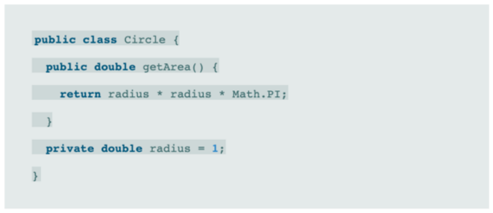
the data field **dateCreated** is returned using the **getDateCreated()** method. This is a reference to a Date object. Through this reference, the content for dateCreated can be changed.

For a class to be immutable, it must meet the following requirements:

1. All data fields must be private.
2. There can't be any mutator methods for data fields.
3. No accessor methods can return a reference to a data field that is mutable.

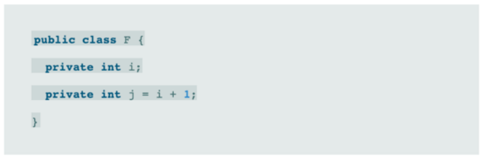
- **Scope of Variables**

The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables in a class are referred to as the class's variables or data fields. A variable defined inside a method is referred to as a local variable. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class. The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first.



```
public class Circle {
  public double getArea() {
    return radius * radius * Math.PI;
  }
  private double radius = 1;
}
```

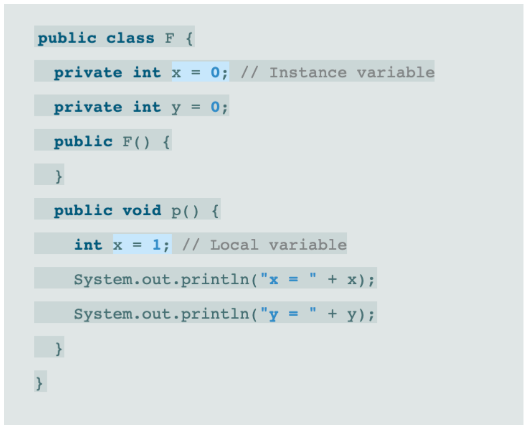
(a) The variable **radius** and method **getArea()** can be declared in any order.



```
public class F {
  private int i;
  private int j = i + 1;
}
```

(b) **i** has to be declared before **j** because **j**'s initial value is dependent on **i**.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different non-nesting blocks. If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is hidden. For example, in the following program, x is defined both as an instance variable and as a local variable in the method:



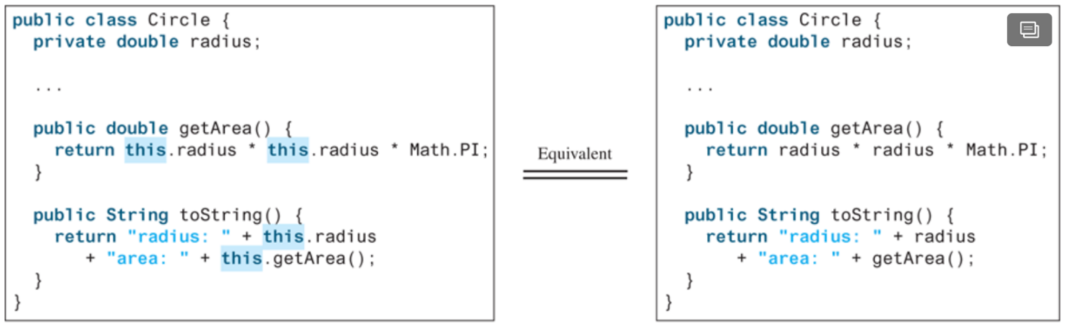
```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;
  public F() {
  }
  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters. We will discuss hidden data fields by method parameters in the next section.

- **The Keyword this**

The keyword this refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.

The **this** *keyword* is the name of a reference that an object can use to refer to itself. You can use the **this** keyword to reference the object's instance members. For example, the following code in (a) uses **this** to reference the object's **radius** and invokes its **getArea()** method explicitly. The **this** reference is normally omitted for brevity as shown in (b). However, the **this** reference is needed to reference a data field hidden by a method or constructor parameter, or to invoke an overloaded constructor.



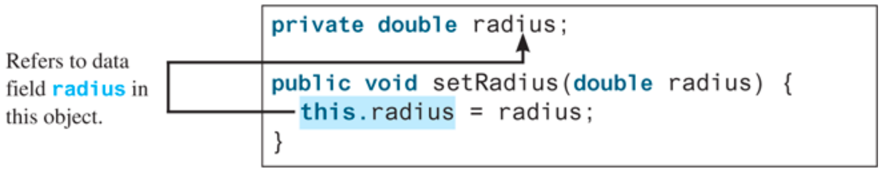
```
public class Circle {
  private double radius;

  ...

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

  public String toString() {
    return "radius: " + this.radius
      + "area: " + this.getArea();
  }
}
```

Equivalent

```
public class Circle {
  private double radius;

  ...

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

  public String toString() {
    return "radius: " + radius
      + "area: " + getArea();
  }
}
```

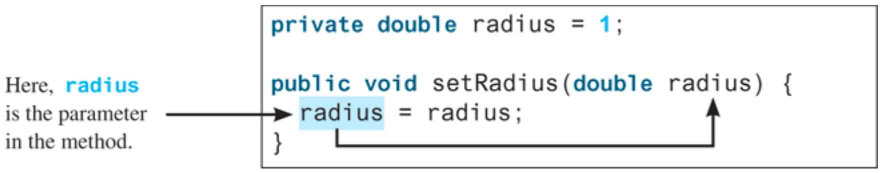
(a) (b)

It is a good practice to use the data field as the parameter name in a setter method or a constructor to make the code easy to read and to avoid creating unnecessary names. In this case, you need to use the **this** keyword to reference the data field in the setter method. For example, the setRadius method can be implemented as shown in (a). It would be wrong if it is implemented as shown in (b).



Refers to data field radius in this object.

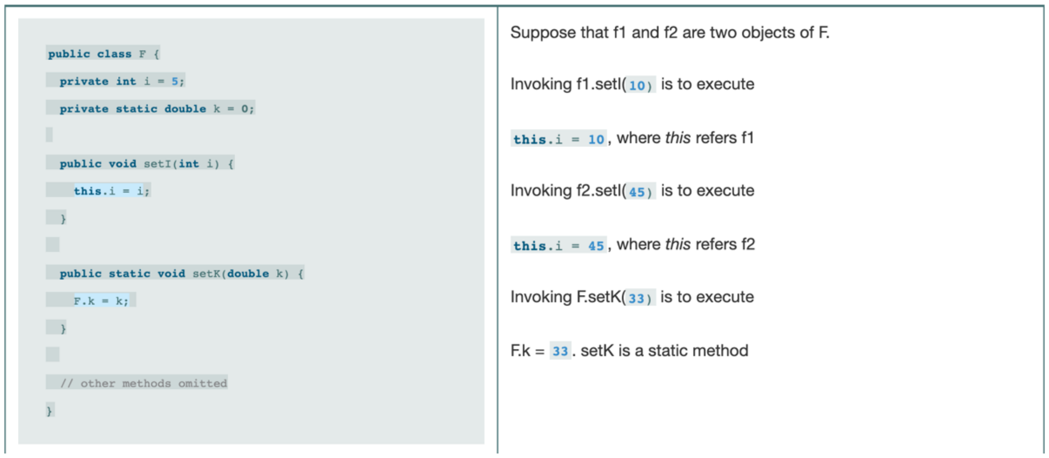
```
private double radius;

public void setRadius(double radius) {
  this.radius = radius;
}
```



Here, radius is the parameter in the method.

```
private double radius = 1;

public void setRadius(double radius) {
  radius = radius;
}
```

The **this** keyword gives us a way to reference the object that invokes an instance method. To invoke **f1.setI(10), this.i = i** is executed, which assigns the value of parameter i to the data field i of this calling object f1. The keyword this refers to the object that invokes the instance method **setI**. The line **F.k = k** means the value in parameter k is assigned to the static data field k of the class, which is shared by all the objects of the class.



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

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

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

  // other methods omitted
}
```

Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute

this.i = 10, where *this* refers f1

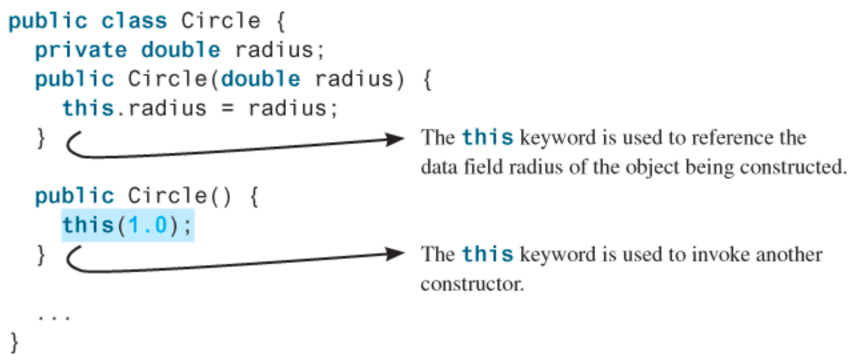
Invoking f2.setI(45) is to execute

this.i = 45, where *this* refers f2

Invoking F.setK(33) is to execute

F.k = 33. setK is a static method

The **this** keyword can be used to invoke another constructor of the same class. For example, you can rewrite the **Circle** class as follows:



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

  public Circle() {
    this(1.0);
  }

  ...
}
```

The this keyword is used to reference the data field radius of the object being constructed.

The this keyword is used to invoke another constructor.

Java requires that the **this(arg-list)** statement appear first in the constructor before any other executable statements.

If a class has multiple constructors, it is better to implement them using **this(arg-list)** as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using this(arg-list). This syntax often simplifies coding and makes the class easier to read and to maintain.