Overview of Object-Oriented Programming Concepts

Object and Class Concepts

Object Concept

An *object* represents an entity in the real world that can be distinctly identified. For example, student, desk, circle, button, person, course, car, employee, department, store, computer, etc…

For instance, an object might represent a particular employee in a company. Each employee object handles the processing and management of data related to that employee.

An object has a unique identity, state, and behaviors.

The *state* of an object consists of a set of *data fields* (*instance variables*, *attributes*, or *properties*) with their current values.

The *behavior* of an object is defined by a set of methods defined in the class from which the object is created.

Class Concept

In OOP, a class is the blueprint (template) that defines a set of real-live objects of the same type, such as students, cars, employees, computers, building, etc.

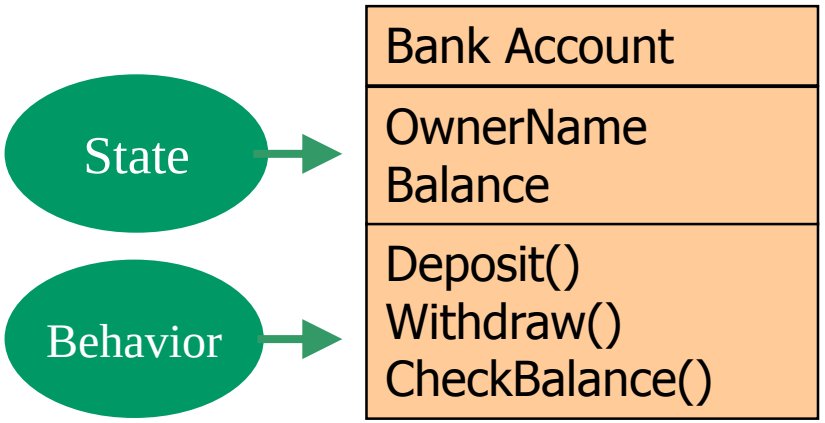
The class uses methods to define the behaviors of its objects, and data fields to represent the state of an object.

In Java, the class may provide a special type of methods, known as constructors, which are used to construct (build/create) objects from the class.

Multiple objects can be created from the same class.

Class Example

A class
(the concept)



Object
(the implementation)

John's Bank Account
Balance: $7,890

Amy's Bank Account
Balance: $298,987

Ed's Bank Account
Balance: $860,883

Multiple objects
from the same class

Memory Representation of Objects

In OOP (e.g., Java), an object is associated with a memory space referenced (pointed at) by the object name.

The memory space is allocated, by the Operating System, when using the **new** operator to create the object.

The memory space allocated for an object holds the values of the data fields (instance variables) of the object. Those are the variables defined in the object's class.

The **constructor** method creates the object (i.e., populates the object's memory space with values).

Constructor Methods (in Java)

In Java, the contractor method (along with the new operator) creates and populates/initializes the object in the memory with the help of the Operating System.

Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing object's space.

A class can have multiple versions of the constructor method, allowing the user to create the object in different ways (and values).

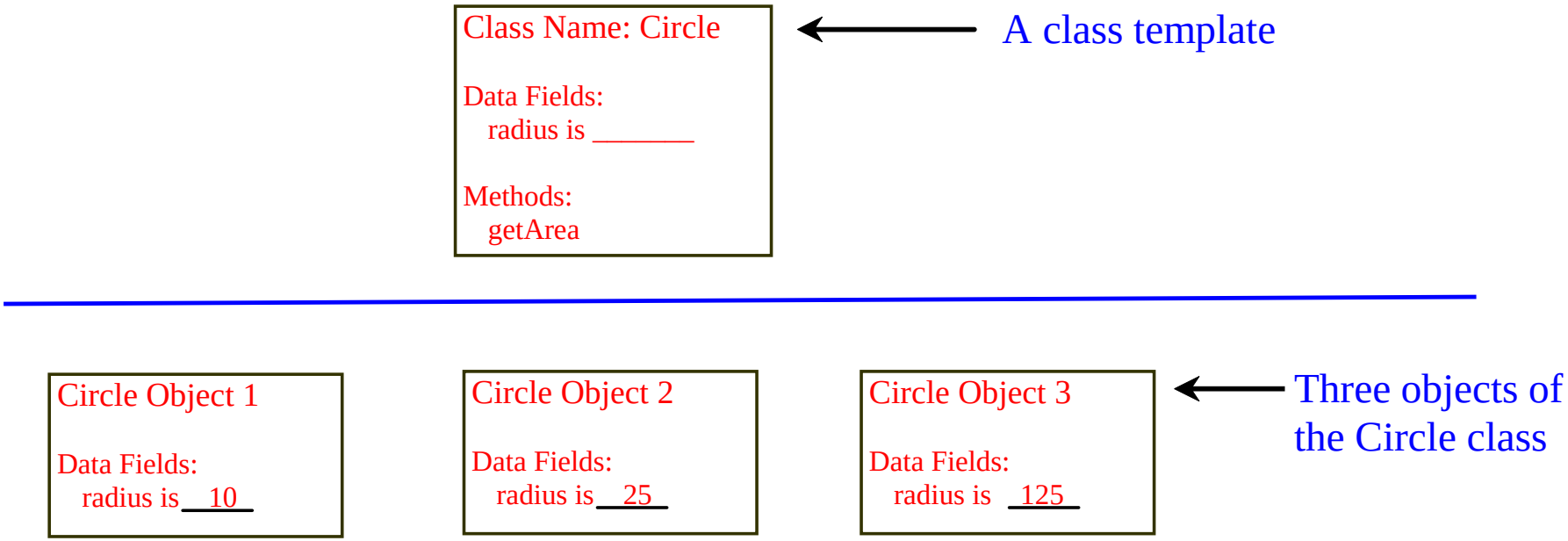
The constructor method must have same name as the class name.

Constructors do not have a return type, not even **`void`**.

A class may be declared without constructors. Thus a *default constructor* is provided by Java.

Writing Classes

A class can contain data fields declarations and method declarations.



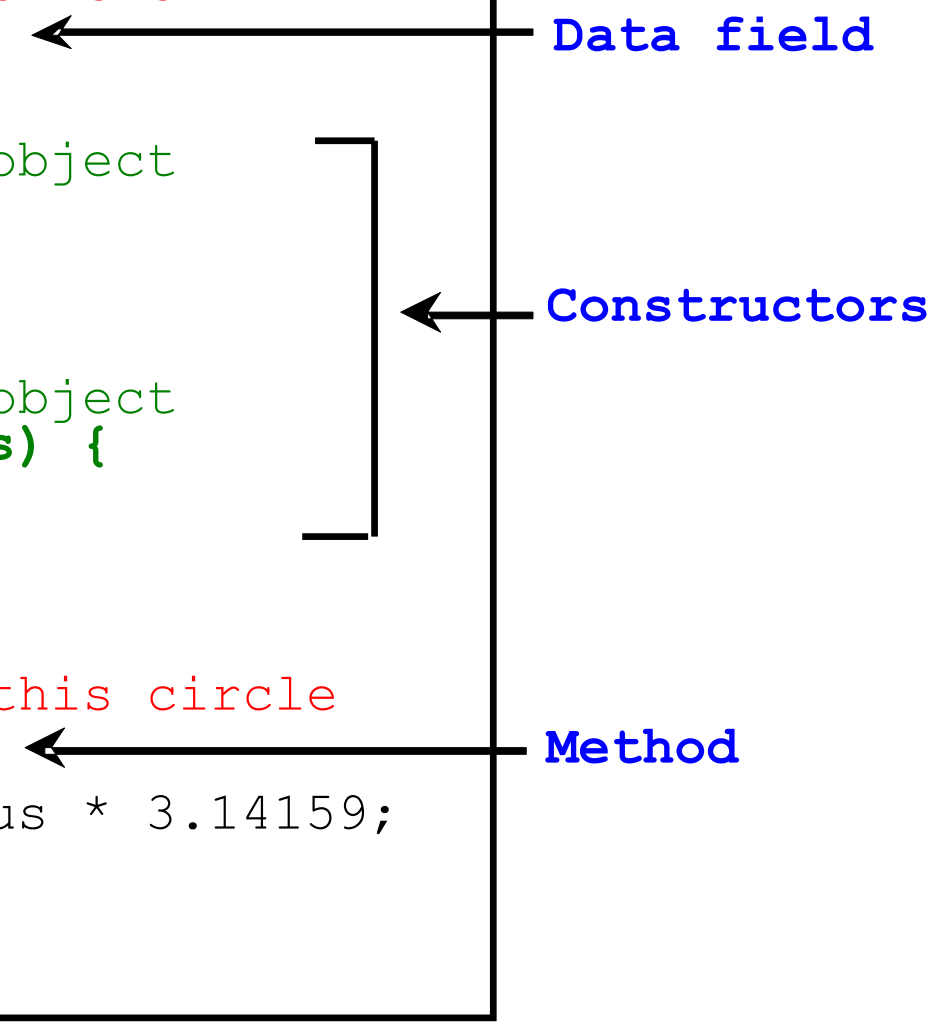
Class Circle Constructors

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

  // Construct a circle object
  Circle() {
  }

  // Construct a circle object
  Circle(double newRadius) {
    radius = newRadius;
  }

  // Return the area of this circle
  double getArea() {
    return radius * radius * 3.14159;
  // other methods
  }
}
```



Creating Circle Objects

To reference an object, assign the object to a reference variable.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Example:

```
Circle circle1, circle2; //declares reference variables
circle1 = new Circle();  //calls first constructor
circle2 = new Circle(25.0); //calls second constructor
```

OR

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(25.0);
```

Accessing the Object

Referencing the object's data:

```
objectRefVar.data
```

```
double radius1 = circle1.radius; //data field access
```

Invoking the object's method:

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

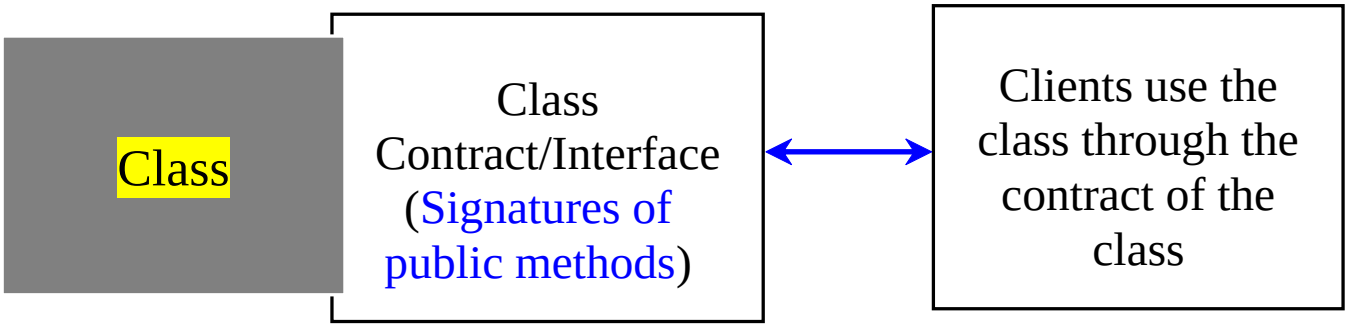
```
double area1 = mcircle1.getArea(); //method access
```

Abstraction and Encapsulation Concepts

Class Abstraction

Abstraction means to separate class implementation details from users of the class. The creator of the class provides a description of the class and lets the user know how the class can be used. User of the class does not need to know how the class is implemented. The details of implementation are encapsulated (hidden) from the user.

Class implementation is like a black box hidden from the clients

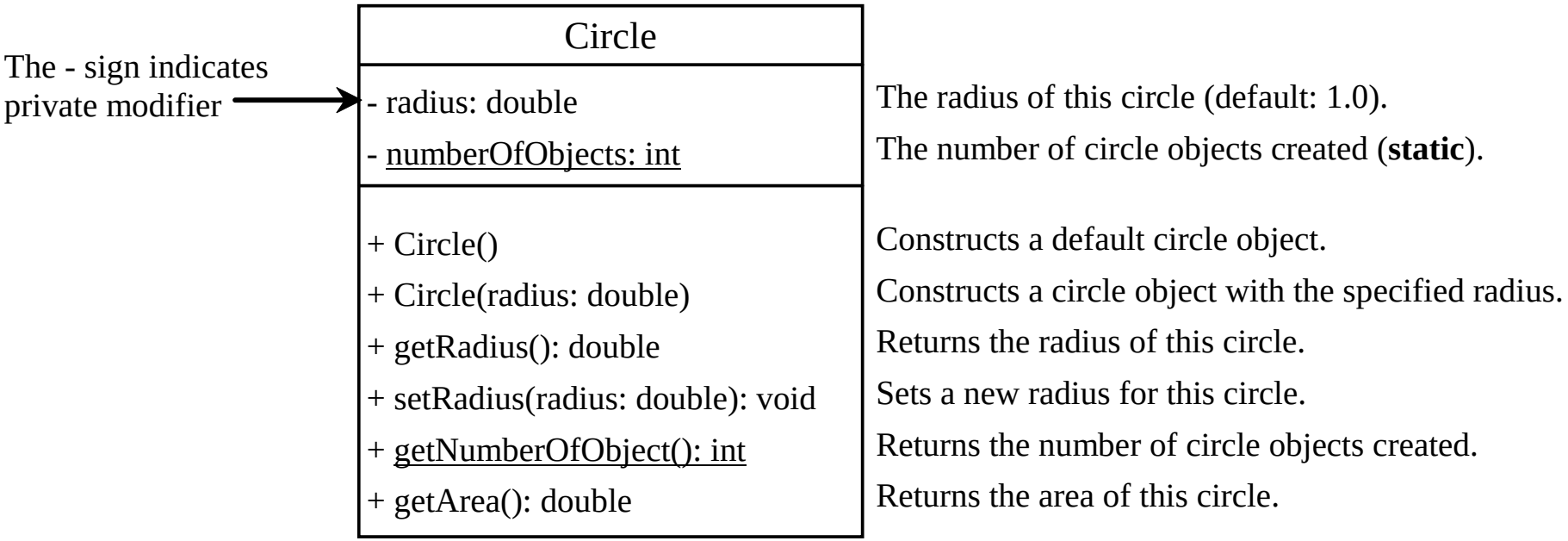


Encapsulation

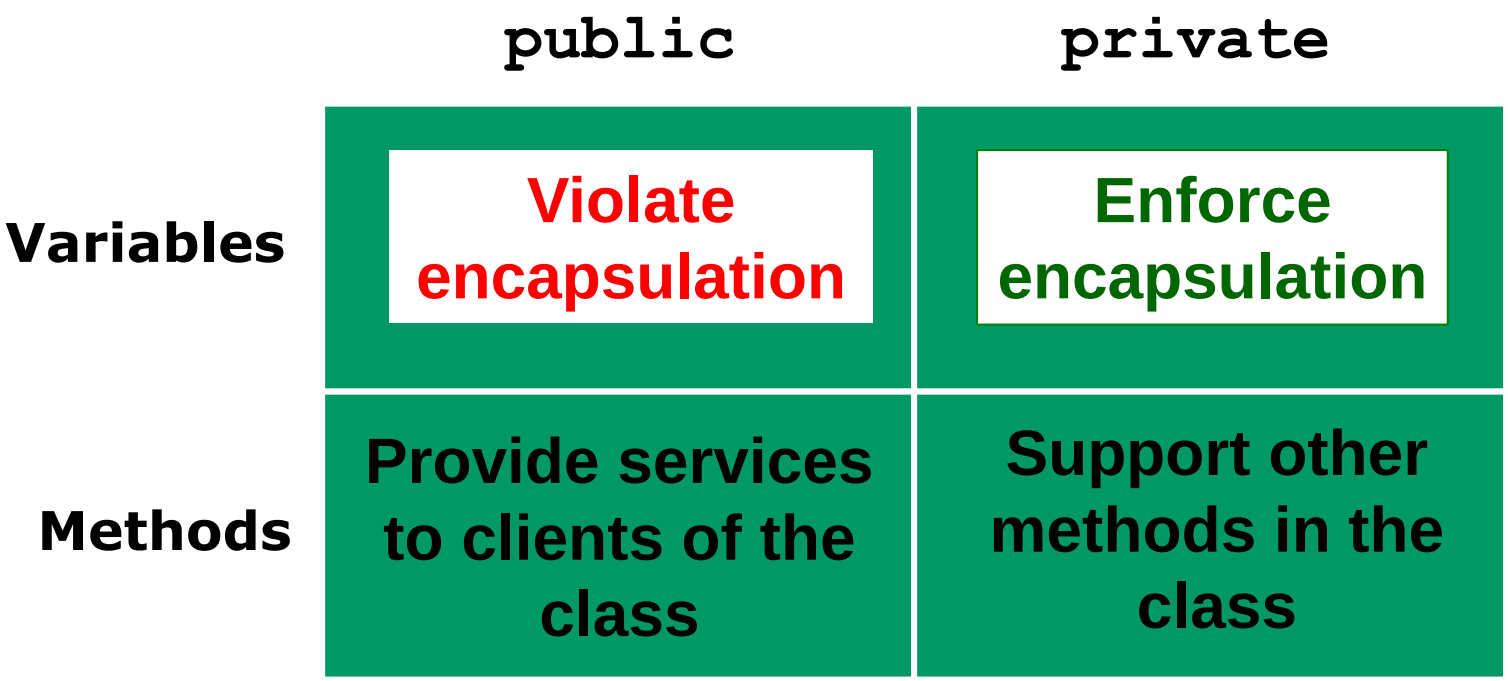
Encapsulation is the idea of hiding the class internal details that are not required by clients/users of the class.

Why? To protect data and to make classes easy to maintain and update.

How? Always use private variables!



Java Visibility Modifiers and Encapsulation



Java Visibility Modifiers

Class members (variables and methods) that are declared with *public visibility* can be referenced/accessed anywhere in the program.

Class members that are declared with *private visibility* can be referenced/accessed only within that class.

Class members declared without a visibility modifier have *default visibility* and can be referenced/accessed by any class in the same package.

Public variables violate encapsulation because they allow class clients to "reach in" and modify the values directly. *Therefore instance variables should not be declared with public visibility.*

Java Visibility Modifiers

Methods that provide the object's services must be declared with *public visibility* so that they can be invoked by clients/users of the class.

Public methods are also called *service methods*.

A method created simply to assist a service method is called a *support method.*

Since support methods are not intended to be called by the class clients, they should be declared with *private visibility.*

Encapsulation Example

```
public class CircleWithPrivateDataFields
{
   private double radius = 1;
   private static int numberOfObjects = 0;

   public CircleWithPrivateDataFields() { numberOfObjects++; }

   public CircleWithPrivateDataFields(double newRadius) {
     radius = newRadius;
     numberOfObjects++;
   }

   public double getRadius() { return radius; }

   public void setRadius(double newRadius) {
     radius = (newRadius >= 0) ? newRadius : 0; //no negative radius
   }

   public static int getNumberOfObjects() {return numberOfObjects; }

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

```
public class TestCircleWithPrivateDataFields {

  public static void main(String[] args) { // Main method

    // Create a Circle with radius 10.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(10.0);

    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);

    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());
  }
}
```

Note: **variable radius cannot be directly accessed. Only through the class methods!**

```
Output:

The area of the circle of radius 10.0 is 314.1592653589793
The area of the circle of radius 11.0 is 380.132711084365
```

Inheritance Concept

Inheritance

Inheritance promotes code reuse to avoid redundancy. That is, developers can define/build new classes from existing classes that share common features. For example, classes Circle, Rectangle, and Triangle can inherits from class Shape (*which represents common features among different shapes*).

A subclass (child/derived) inherits from (builds on or extends) a superclass (parent/base). The subclass may:

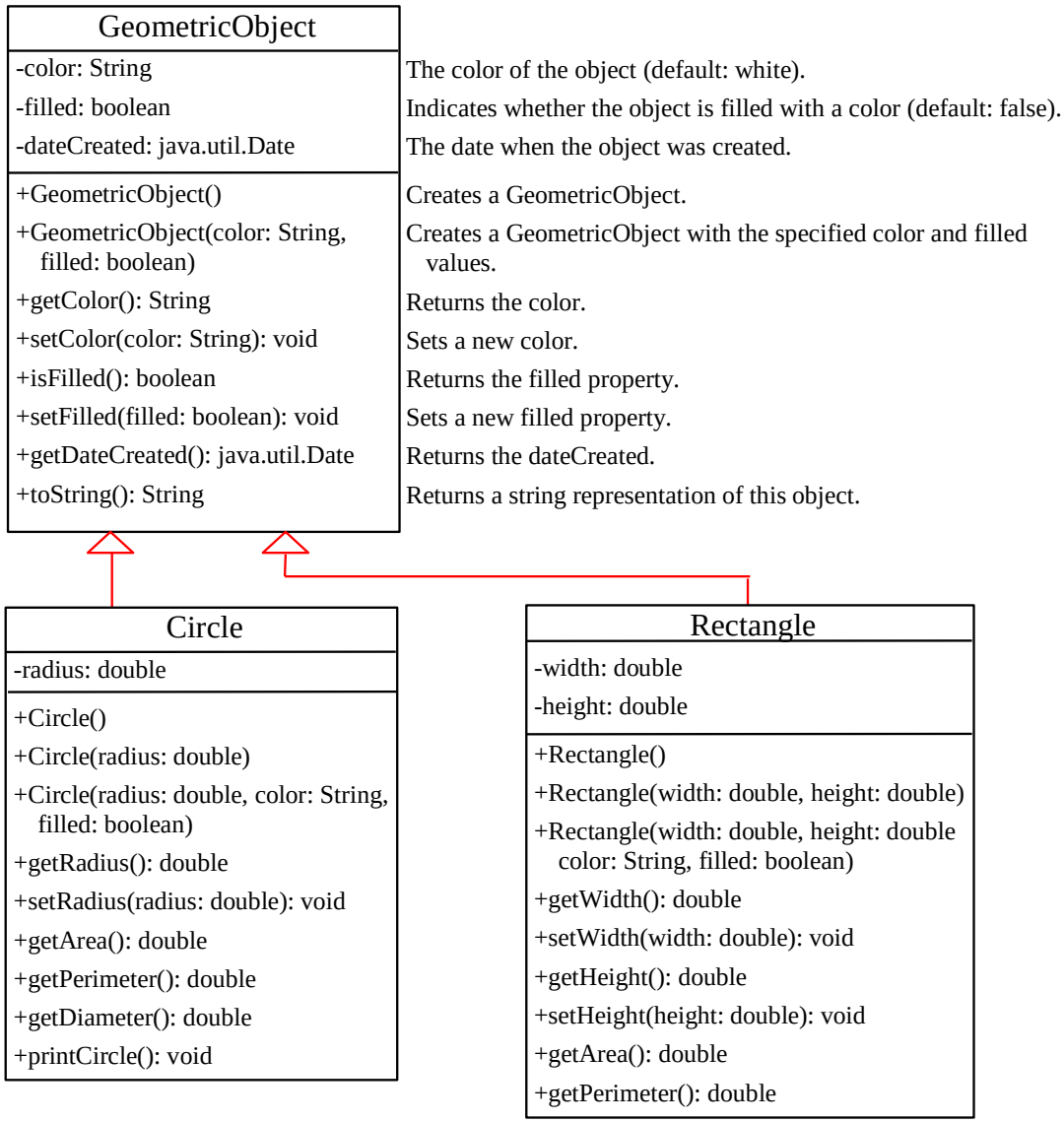
Add new properties; Add new methods; and/or Override the methods of the superclass

Private content is NOT directly accessible to the subclass, only via methods (if provided).

The superclass of all Java classes is the java.lang.Object class.

Java syntax: **`Public class Student extends Person`**

Java Inheritance Example



Calling Superclass Methods

To access parent class methods, one can rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle()
{
   System.out.println("The circle is created " +
   super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends GeometricObject

{
  // Other methods in the class are omitted
  // Override method toString defined in GeometricObject

  @Override
  public String toString()
  {
     return super.toString() + "\nradius is " + radius;
  }

}
```

NOTE

A class method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class.

If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i * 2);
  }
}
```

Note that you may overload and override a method at the same time.

```
// This method overloads the method in B
public void p(int i) {
      System.out.println(i * i);
   }
```

Polymorphism Concept

Polymorphism

Polymorphism is the ability to use an object of the subclass anywhere we expect an object of the superclass because an object of the subclass is an object of the superclass, **but not vice versa**.

OR

Polymorphism is that a variable of a supertype can refer to a subtype object. A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.

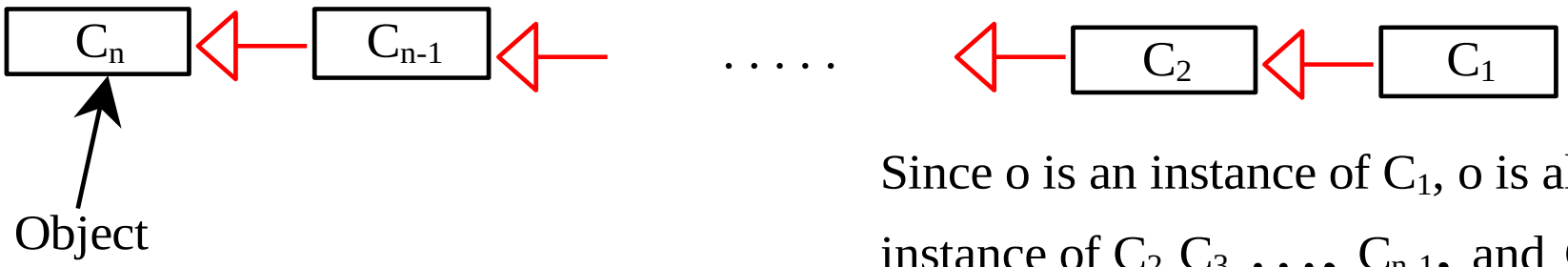
Therefore, **Circle** is a *subtype* of **GeometricObject** and **GeometricObject** is a *supertype* for class **Circle**.

Dynamic Binding

Dynamic binding is mapping method calls to methods. It works as follows:

Suppose an object o is an instance of class C1.

Class C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. If object o invokes a method p, the JVM searches the implementation for the method p in classes C_1, C_2, ..., C_{n-1} and C_n, in this order, until method p is found. Once method p implementation is found, the search stops and the first-found implementation is excuted.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . ., C_{n-1}, and C_n

Method Matching vs. Binding

Matching a method signature (at compile time) and binding a method implementation (at runtime) are two different things.

The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.

A method may be implemented in several subclasses. The JVM dynamically binds an implementation of the method at runtime.

The Big Picture

```
public class PolymorphismDemo {
  public static void main(String[] args)
{
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}
// ===================================
class GraduateStudent extends Student
{
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method **m** takes a parameter of the Object type. You can invoke it with any object type.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When method m(Object x) is executed, the argument x's toString() method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Since classes GraduateStudent, Student, Person, and Object have their own implementation of method toString(). Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Implicit Casting of Objects

Casting (type conversion) can be used to convert an object of one class type to another within an inheritance hierarchy.

On the previous slide, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type in method **m**. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student() is known as implicit casting. It is legal because an instance of class Student is automatically an instance of class Object.

Explicit Casting of Objects

Suppose you want to assign the object reference **o** (**o** is of type *Object*) to a variable of the *Student* type using the following statement:

```
Object o = new Object();
Student b = o; //compile error. O can be anything!
```

Question: Why does `Object o = new Student();` work but `Student b = o;` doesn't?

This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though we can see that o is really a Student object, the compiler is not so clever enough to know it. To tell the compiler that o is a Student object, use explicit casting:

```
Student b = (Student)o; //Explicit casting of Student
                        //type to Object type
```

Hint

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for classes Apple and Orange.

An apple is a fruit, so you can always safely assign an object of class Apple to a variable of type Fruit. However, a fruit is not necessarily always an apple, so you have to use explicit casting to assign an object of class Fruit to a variable of Apple.

```
Fruit aFruit;
Apple redApple;
Orange floridaOrange;
...
aFruit = redApple;
...
aFruit = floridaOrange;
```

```
Fruit myFruit;

Apple  x = (Apple)myFruit;
...
Orange y = (Orange)myFruit;
...
Banana z = (Banana)myFruit;
```

Java `instanceof` Operator

Use the **`instanceof`** operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
... // Some lines of code
// Perform casting if myObject is an instance of Circle
if (myObject instanceof Circle)
{
  System.out.println("The circle diameter is " +
                     ((Circle)myObject).getDiameter());
  ... // Some lines of code
}
... // Some lines of code
```

Java Method `equals()` in Class Object

Method **`equals()`**, in class Object, compares the contents of two objects. The default implementation of the method in the Object class is as follows:

```
public boolean equals(Object obj)
{
  return this == obj; //same memory space
}
```

For example, the equals() method is overridden in class Circle.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

Operator `==` vs. Method `equals()`

The comparison operator `==` is used for comparing two primitive data type values or to determine whether two objects have the same reference (i.e., memory address).

Method equals() is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects.

Therefore, The == operator is stronger than method equals() in that the == operator checks whether or not the two reference variables refer to the same object in the memory.

End of Review