Revision (OOP)

WIA1002/WIB1002 : Data Structure

Part 1

Check point 1

1. Name an object.
2. What can the object do?
3. Name a different object.
4. Does this 2nd object have the same function(s) as the initially named object?

Class & Object

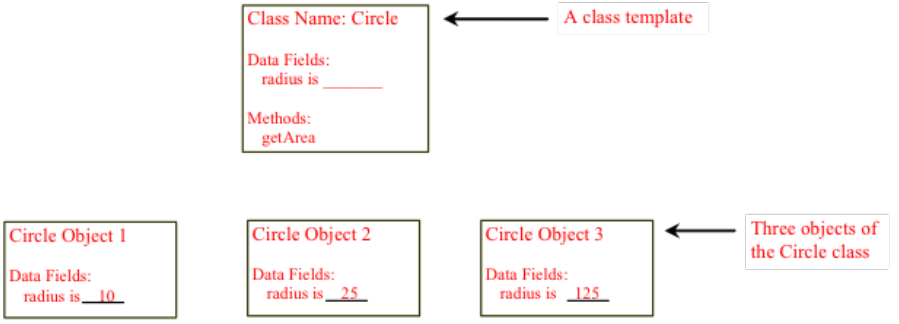
- Object : represents an entity in the real world that can be distinctly identified (for example, a car, apple, ball, chair, computer)
- Object has
  - unique identity
  - state (a set of variables/data fields/properties with their current values)
  - behaviors (a set of methods)

Class & Object

- Class
  - a *construct* that defines objects of the same type.
  - is a template, blueprint, or *contract* that defines what an object's data fields and methods will be.

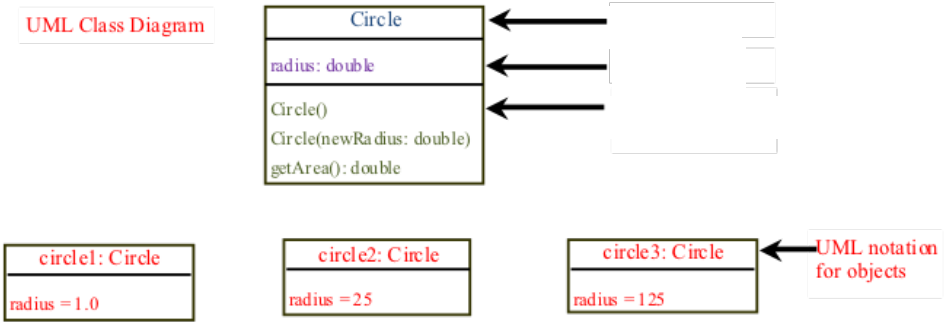
- An object is an instance of a class. Can create many objects/instances of a class.

Objects



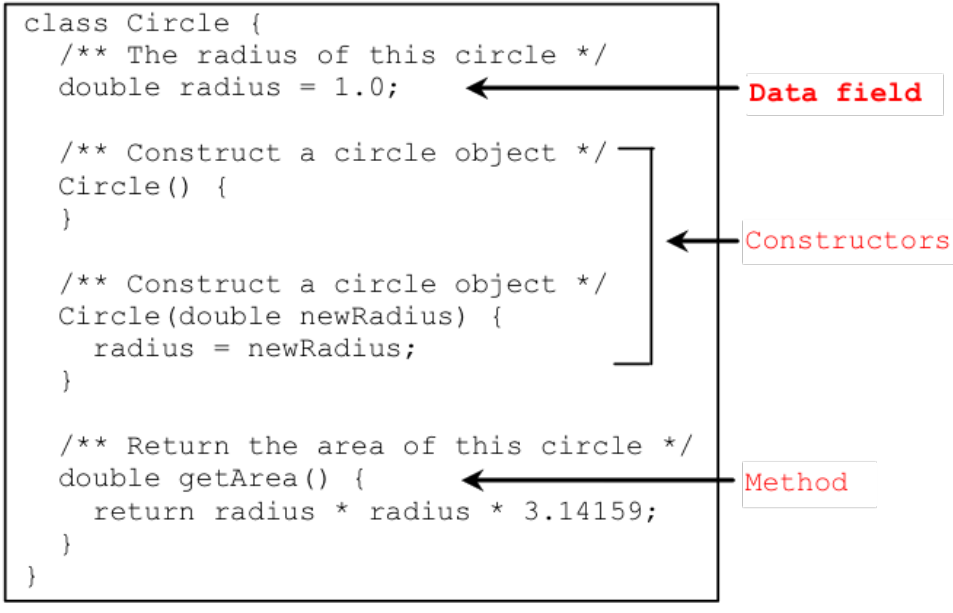
An **object** has both a state and behavior. The state defines the object, and the behavior defines what the object does.

UML Class Diagram



Question: Declare the class, properties and methods for the UML diagram above.

Classes


class Circle {
/** The radius of this circle */
double radius = 1.0;
/** Construct a circle object */
Circle() {
}
/** Construct a circle object */
Circle(double newRadius) {
radius = newRadius;
}
/** Return the area of this circle */
double getArea() {
return radius * radius * 3.14159;
}
}
Data field
Constructors
Method

Constructors

```
Circle() {
}

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

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

Constructors, cont.

- Constructor
- A constructor with no parameter is referred to as a *no-arg constructor*.
- must have the **same name as** the **class** itself.
- **do not have a return type**—not even void.
- is **invoked using the new operator** when an object is created. Constructors play the role of initializing objects.

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*.

Check point 2

- What is the difference between constructors and methods?
- Can a class have more than one constructors?
- What are constructors used for? and how?
- What operator is used to invoke a constructor?
- When will a class have a default constructor?

Declaring Object Reference Variables

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

To declare a reference variable, use the syntax:

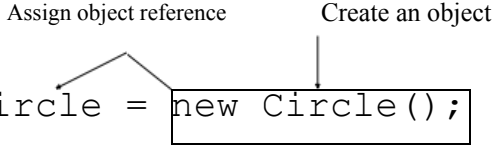
```
ClassName objectRefVar;
```

Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

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



Example:

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

Accessing Object's Members

- Referencing the object's data:

  `objectRefVar.data`

  e.g., `myCircle.radius`

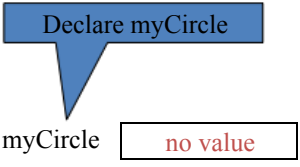
- Invoking the object's method:

  `objectRefVar.methodName(arguments)`

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

animation

Trace Code



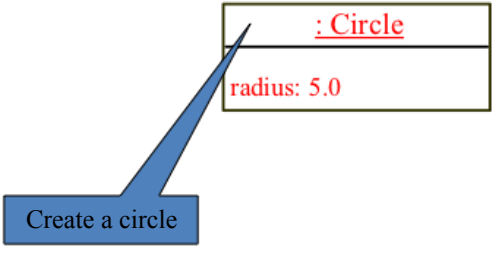

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

animation

Trace Code, cont.

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

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

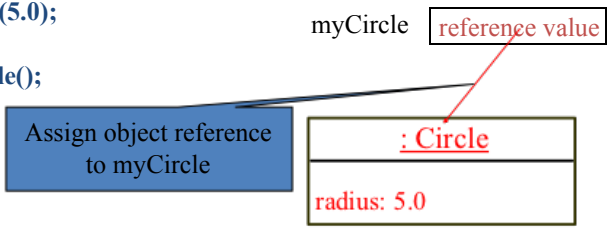
myCircle | no value



animation

Trace Code, cont.

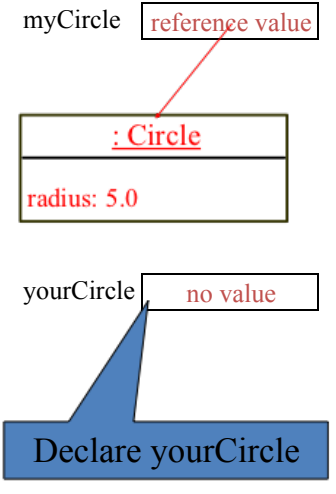
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

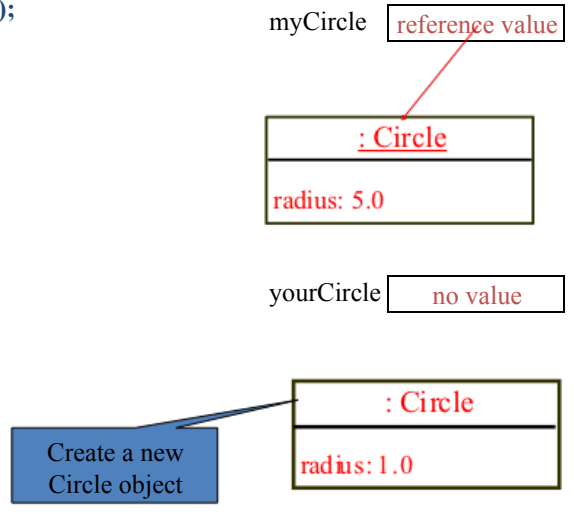
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

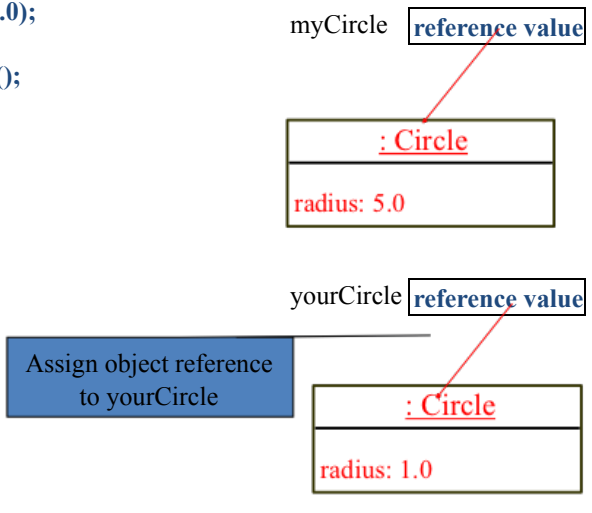
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



animation

Trace Code, cont.

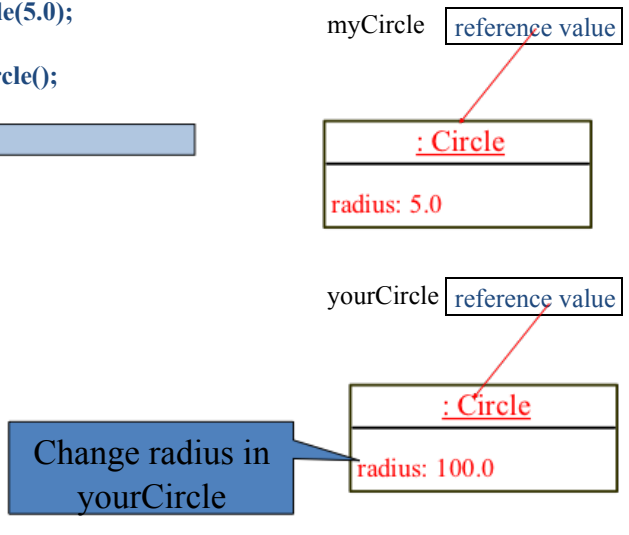
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



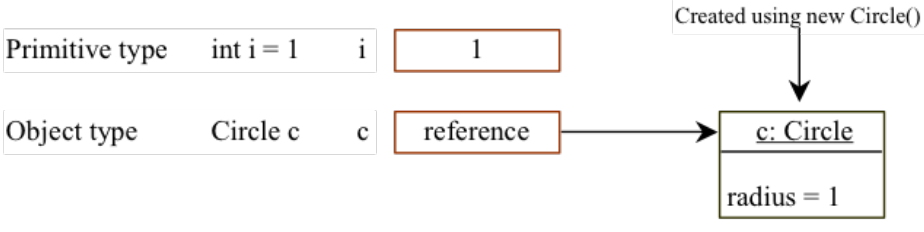
animation

Trace Code, cont.

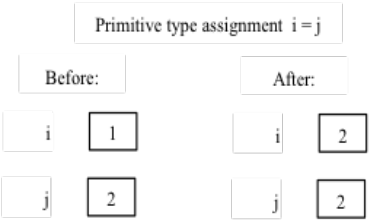
```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```

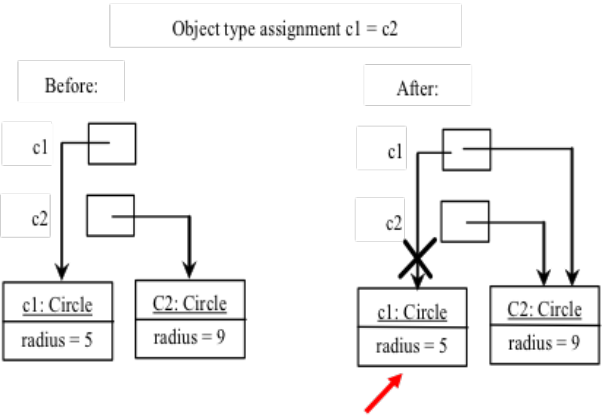


Differences between Variables of Primitive Data Types and Object Types



Copying Variables of Primitive Data Types and Object Types





This object which is previously referenced by c1 is no longer referenced and and is considered as garbage that will be collected automatically by JVM.

Visibility Modifiers and Accessor/Mutator Methods

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

- `public`

The class, data, or method is visible to any class in any package.

- `private`

The data or methods can be accessed only by the declaring class.

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

```
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.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

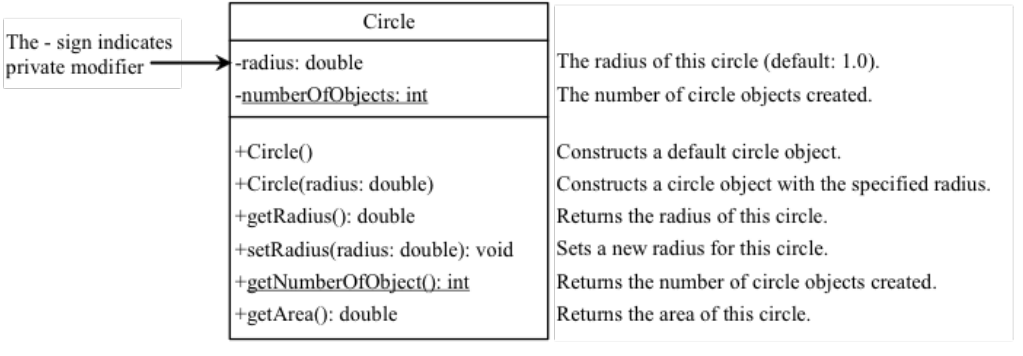
```
package p2;

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

Why Data Fields Should Be private?

To protect data.

To make class easy to maintain.

Example of Data Field Encapsulation



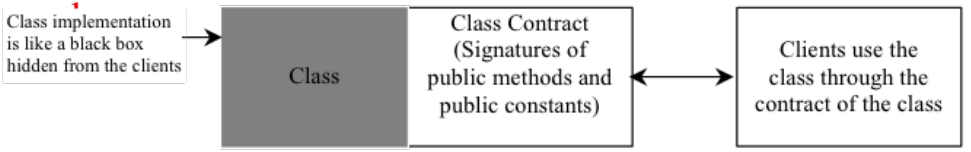
CircleWithPrivateDataFields

TestCircleWithPrivateDataFields

Part 2

Class Abstraction and Encapsulation

- 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 who uses the



Part 3

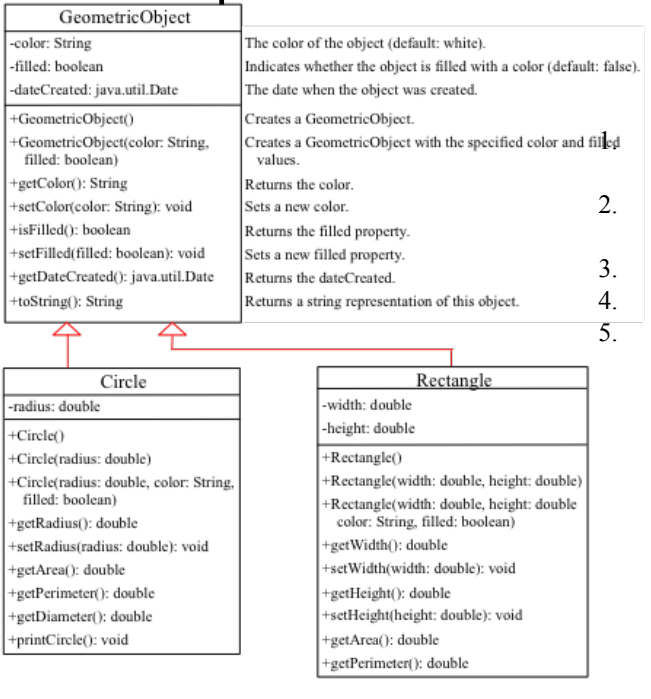
Inheritance

Circles, rectangles, triangles..do they share common features?

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy?

The answer is to use **inheritance**.

Superclasses and Subclasses



Q:

1. **Is subclass a subset of superclass?**
2. **Private accessible outside class?**
3. **Extensible?**
4. **Multiple inheritance?**
5. **Superclass constructor inherited?**

Calling superclass constructor

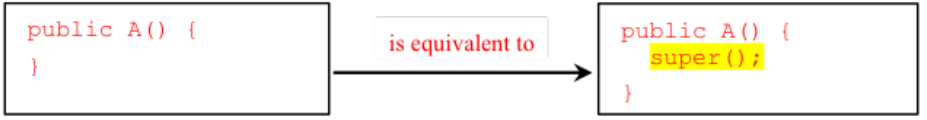
- Syntax :

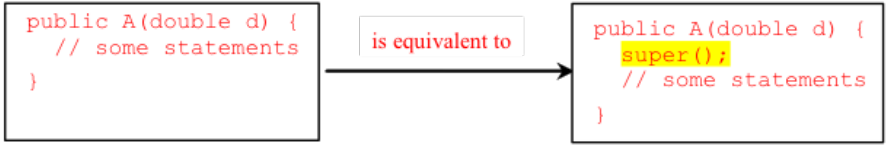
  super() or **super(parameters)**

```
public Circle(double radius, String color, boolean filled){
 super(color, filled);
 this.radius = radius;
}
```

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,





- What's the printout of running class C? In both (a) and (b).

```
class A {
  public A() {
    System.out.println(
      "A's no-arg constructor is invoked");
  }
}

class B extends A {
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(a)

```
class A {
  public A(int x) {
  }
}

class B extends A {
  public B() {
  }
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(b)

(a) A's no-arg constructor is invoked

(b) error – no A() constructor

Overriding vs. Overloading

- Overloading : define multiple methods , same name but different parameter list; can be in same or different classes related by inheritance

- Overriding : provide new implementation for a method in the subclass; different classes related by inheritance; same method name, same parameter list and return type

Overriding vs. Overloading

```java
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);
  }
}
```

(a)

```java
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);
  }
}
```

(b)

Output (a) : 10
a.p(10) & a.p(10.0) both
invoke p(double i) defined in class A

Output (b) : a.p(10) – p(int i) - 10
a.p(10.0) – p(double i) – 20.0

Polymorphism

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

```
e.g 1 :
GeometricObject geoObj = new Circle();
```

```
e.g 2 :
Fruit f = new Apple();
```

Generic Programming, polymorphism, and dynamic binding

```
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";
 }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Output :

Student
Student
Person
Java.lang.Object@39274

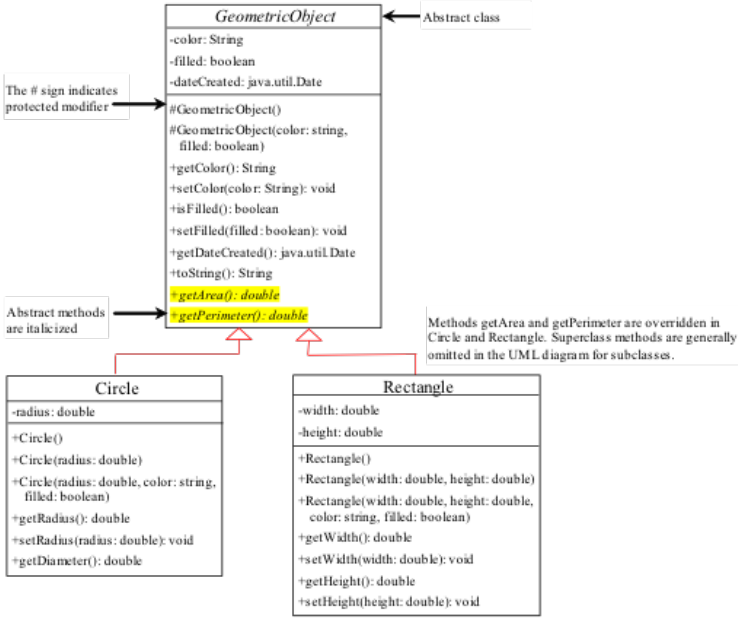
Visibility/Accessibility Modifiers Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Visibility Modifiers

Part 4

Abstract Classes and Abstract Methods



GeometricObject

Circle

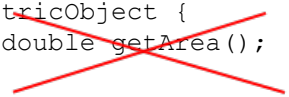
Rectangle

TestGeometricObject

1. abstract method in abstract class

An abstract method cannot be contained in a nonabstract class.

```
public class GeometricObject {
  public abstract double getArea();
}
```



If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

2. object cannot be created from abstract class

An abstract class cannot be instantiated using the new operator

```
GeometricObject geoObj = new GeometricObject();
```

but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

3. abstract class without abstract method

A class that contains abstract methods must be abstract.

```
public abstract class GeometricObject {
  public abstract double getArea();
}
```

However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

4. superclass of abstract class may be concrete

A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

```
public class Object {}
public abstract class GeometricObject {}
```

5. concrete method overridden to be abstract

A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

```
public abstract class
  GeometricObject{

   public String someMethod();

}

public abstract class Circle
  extends GeometricObject {

 public abstract String
  someMethod();

}
```

6. abstract class as type

You cannot create an instance/object from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new GeometricObject[10];
geo[0] = new Circle();
geo[1] = new Rectangle();
```

What is an interface? Why is an interface useful?

- A classlike construct that contains only constants and abstract methods.
- Similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.
- Cannot create an instance using the new operator

Define an Interface

To distinguish an interface from a class, Java uses the following **syntax** to define an interface:

```
public interface InterfaceName {
 constant declarations;
 method signatures;
}
```

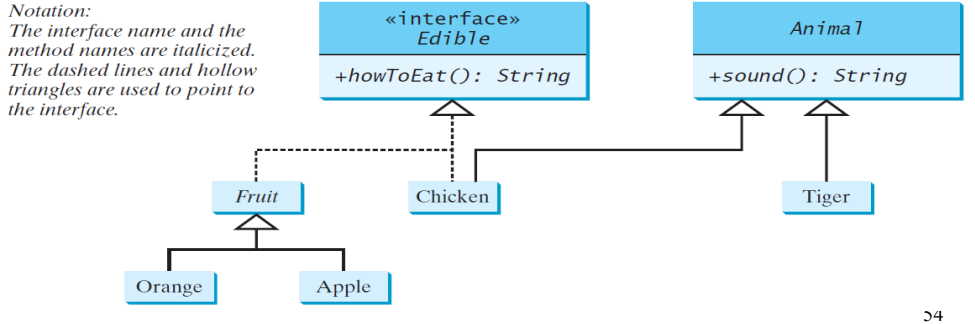
Example:

```
public interface Edible {
 /** Describe how to eat */
 public abstract String howToEat();
}
```

Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).

Edible TestEdible Run



```java
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

LISTING 13.7 TestEdible.java

```java
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++) {
      if (objects[i] instanceof Edible)
        System.out.println(((Edible)objects[i]).howToEat());

      if (objects[i] instanceof Animal) {
        System.out.println(((Animal)objects[i]).sound());
      }
    }
  }
}

abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}

class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}

abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

Question: What is the output?

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

- Java allows only *single inheritance* for class extension but allows *multiple extensions* for interfaces.

```
public class NewClass extends BaseClass
  implements Interface1, ...., InterfaceN {

}
```

- An interface can inherit other interfaces using extends keyword. The extended interface is called subinterface.

```
public interface NewInterface extends
  Interface1, ...., InterfaceN {
   //constants and abstract methods
}
```

Extend vs Implement

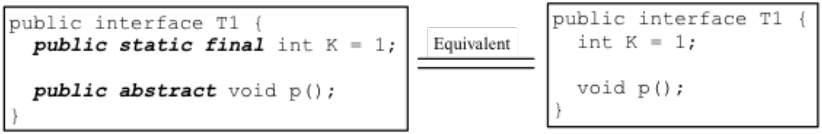
A Java class may extend one class and implement multiple interfaces.

An interface may extend any number of interfaces; however an interface may not implement an interface.

A **class** that **implements** an **interface must implement all** of the **methods** described **in the interface**, or be an abstract class.

Omitting Modifiers in Interfaces

All data fields are *public static final* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:



A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Abstract class

```
abstract class Animal {
  public abstract String howToEat();
}
```

Two subclasses of `Animal` are defined as follows:

```
class Chicken extends Animal {
  @Override
  public String howToEat() {
    return "Fry it";
  }
}

class Duck extends Animal {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  animal.howToEat();
}
```

Interface

```
public static void main(String[] args) {
  Edible stuff = new Chicken();
  eat(stuff);

  stuff = new Duck();
  eat(stuff);

  stuff = new Broccoli();
  eat(stuff);
}

  public static void eat(Edible stuff) {
    stuff.howToEat();
  }

  interface Edible {
    public String howToEat();
  }

  class Chicken implements Edible {
    @Override
    public String howToEat() {
      return "Fry it";
    }
  }

  class Duck implements Edible {
    @Override
    public String howToEat() {
      return "Roast it";
    }
  }

  class Broccoli implements Edible {
    @Override
    public String howToEat() {
      return "Stir-fry it";
    }
  }
```

Check point 3

1. How to do you create an interface called Walking having a howToWalk()?
2. If A is an abstract class. Can you create an instance using new A() ? (NO)
3. If A is an abstract class. Can you declare a reference variable x with type A like this?

   A x; (Yes)

- Which is a correct interface?

```
interface A {
  void print() { };
}
```

(a)

```
abstract interface A extends I1, I2 {
  abstract void print() { };
}
```

(b)

```
abstract interface A {
  print();
}
```

(c)

```
interface A {
  void print();
}
```

(d)

Correct answer : (d)

References

- Part 1: Chapter 9 Objects and Classes, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
- Part 2: Chapter 10 Thinking in Objects, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
- Part 3: Chapter 11 Inheritance and Polymorphism, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015