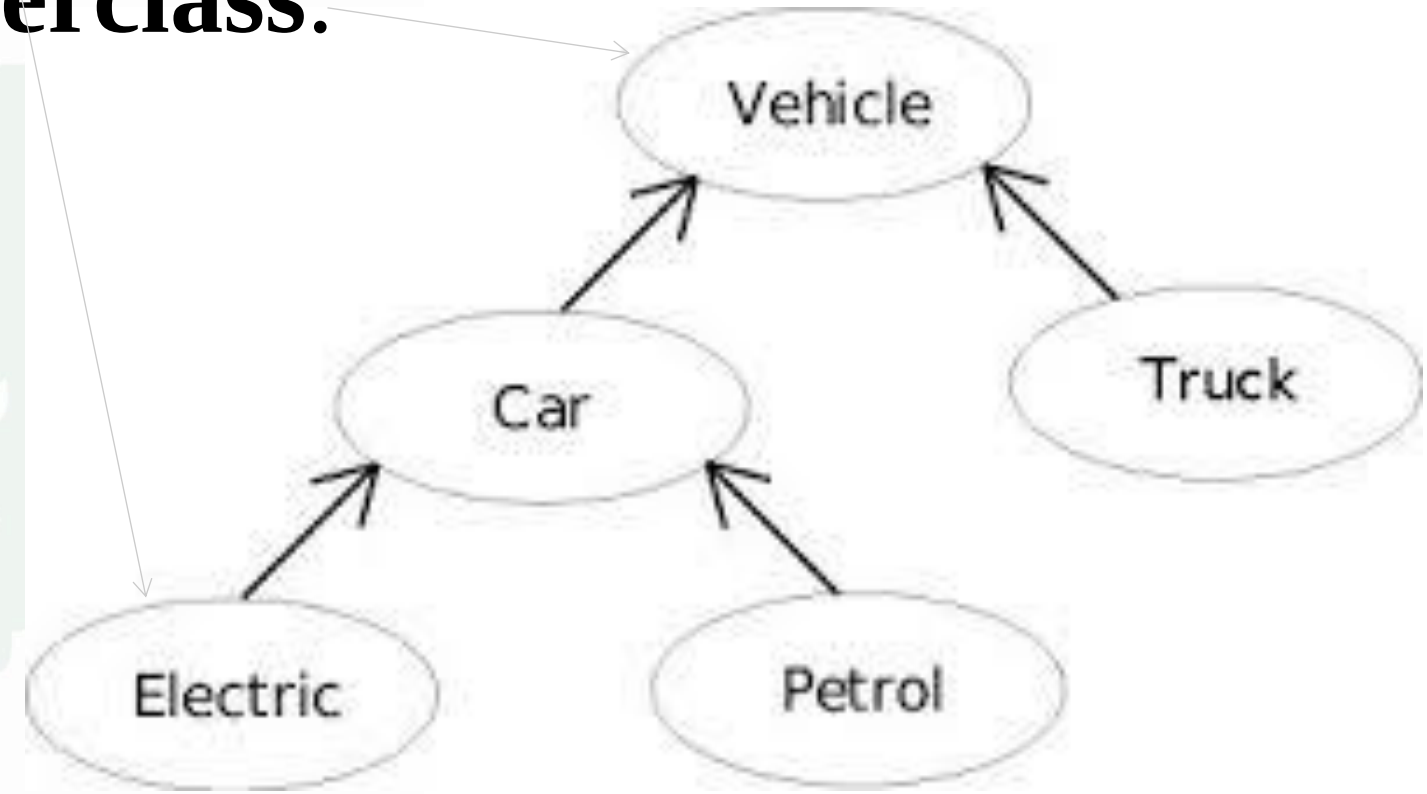
BIRZEIT UNIVERSITY

Chapter 11 Inheritance and Polymorphism

Creating classes from other classes!

Inheritance cont.

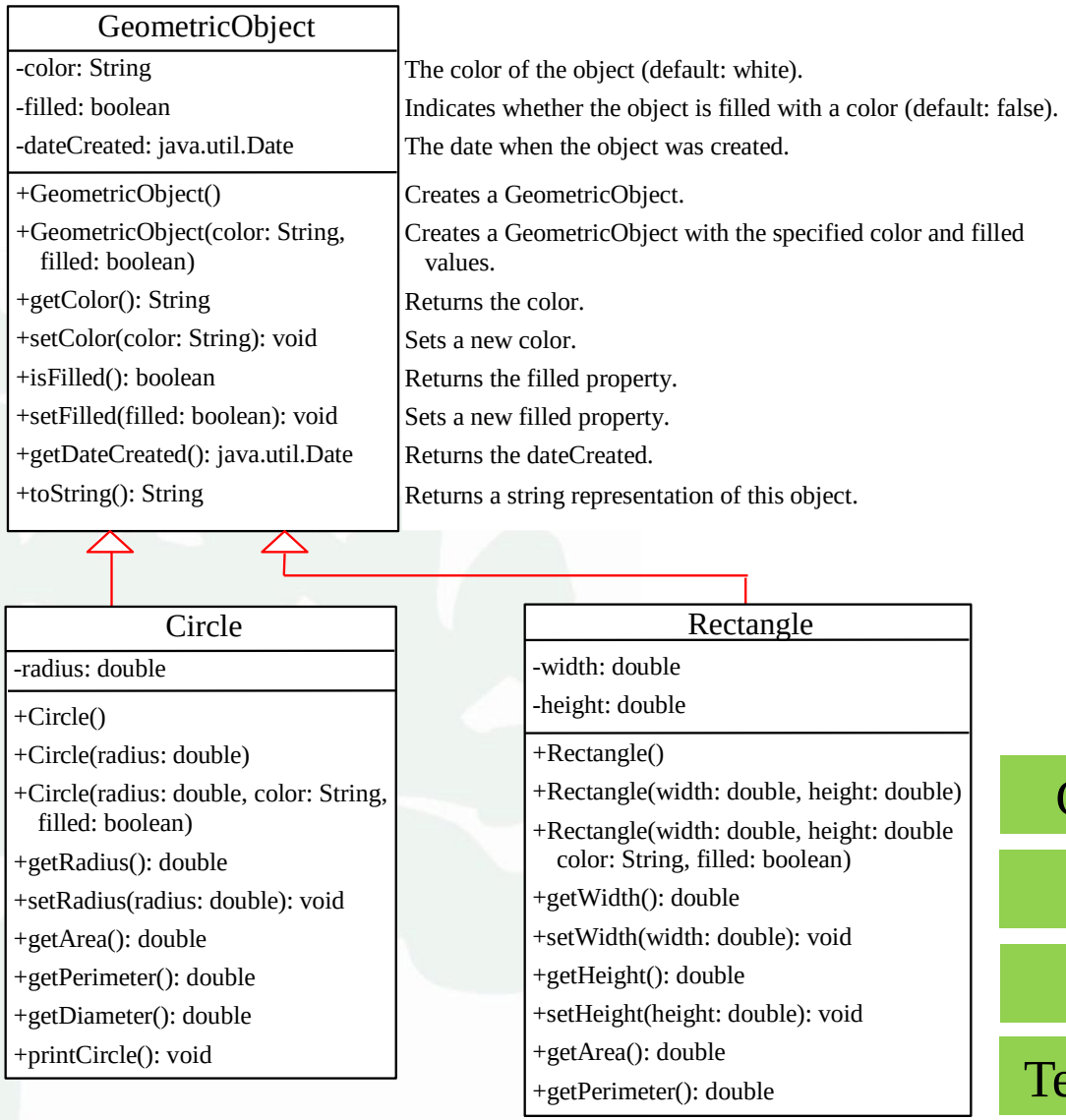
- **Subclasses**: a subclass may inherit the structure and behaviour of it's **superclass**.



Inheritance

- You use a class to model objects of the same type.
- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes.
- You can define a specialized class that extends the generalized class.
- The specialized classes inherit the properties and methods from the general class.

Superclasses and Subclasses



GeometricObject

Circle

Rectangle

TestCircleRectangle

Run

A subclass inherits accessible data fields and methods from its superclass

Does it inherit class constructors?!!

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

What if there is no-arg constructor defined in the super class?!!!

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,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Caution

You must use the keyword `super` to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor's name in a subclass causes a syntax error.

Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

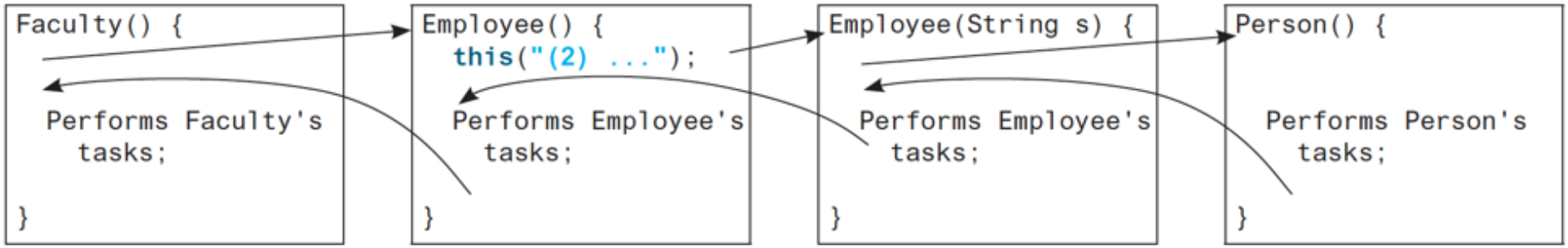
You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Output




```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```

What is the output?

```
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();
  }
}
```

What is the Output?

```
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();
  }
}
```

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Design Guide

If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

Defining a Subclass

A subclass inherits from a superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Calling Superclass Methods

You could 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 are omitted

  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

To override a method, the method must be defined in the subclass using the same signature as in its superclass.

NOTE

An instance 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.

NOTE

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

The hidden static methods can be invoked using the syntax SuperClassName.staticMethodName.

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);
  }
}
```

```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);
  }
}
```

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class, or in different classes related by inheritance.
- Overridden methods have the same signature; overloaded methods have the same name but different parameter lists.

```java
public class Circle {
  private double radius;

  public Circle(double radius) {
    radius = radius;
  }

  public double getRadius() {
    return radius;
  }

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

class B extends Circle {
  private double length;

  B(double radius, double length) {
    Circle(radius);
    length = length;
  }

  @Override
  public double getArea() {
    return getArea() * length;
  }
}
```

Problems?!

This annotation denotes that the annotated method is required to override a method in its superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error

The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```

The toString() method in Object

The toString() method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object (memory add.).

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.

Polymorphism

Three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism

Polymorphism means 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, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

PolymorphismDemo Run

```
tatic void main(String[] args) {
GraduateStudent());
Student());
Person());
Object());


tatic void m(Object x) {
.out.println(x.toString());


uateStudent extends Student {


ent extends Person {
tring toString() {
 "Student";


on extends Object {
tring toString() {
 "Person";
```

DynamicBindingDemo

Run

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

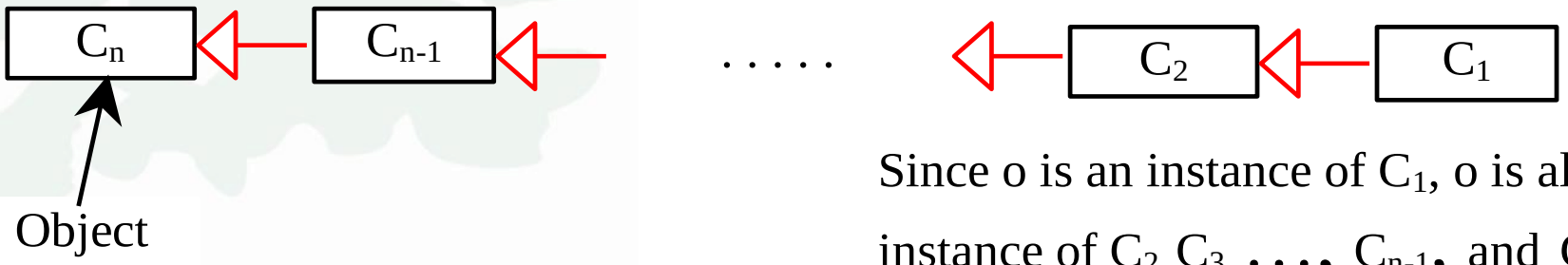
An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the 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. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

```
Student
Student
Person
java.lang.Object@130c19b
```

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where 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. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



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 and binding a method implementation are two issues. 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 Java Virtual Machine dynamically binds the implementation of the method at runtime.

Generic Programming

```
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.

What's wrong?

```
public class Test {
  public static void main(String[] args) {
    Integer[] list1 = {12, 24, 55, 1};
    Double[] list2 = {12.4, 24.0, 55.2, 1.0};
    int[] list3 = {1, 2, 3};
    printArray(list1);
    printAr|ray(list2);
    printArray(list3);
  }

  public static void printArray(Object[] list) {
    for (Object o: list)
      System.out.print(o + " ");
    System.out.println();
  }
}
```

What's the output?!

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  private String getInfo() {
    return "Student";
  }
}

class Person {
  private String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

What's the Output?

```
public class Test {
  public static void main(String[] args) {
    A a = new A(3);
  }
}

class A extends B {
  public A(int t) {
    System.out.println("A's constructor is invoked");
  }
}

class B {
  public B() {
    System.out.println("B's constructor is invoked");
  }
}
```

What's the Output

BIRZEIT UNIVERSITY

```
public class Test {
  public static void main(String[] args) {
    new A();
    new B();
  }
}
```

```
class A {
  int i = 7;

  public A() {
    setI(20);
    System.out.println("i from A is " + i);
  }

  public void setI(int i) {
    this.i = 2 * i;
  }
}

class B extends A {
  public B() {
    System.out.println("i from B is " + i);
  }

  public void setI(int i) {
    this.i = 3 * i;
  }
}
```

Casting Objects

☞ **Casting** can also be used to convert an object of one class type to another **within an inheritance hierarchy**.

m(new Student());

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

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

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

Why Casting is Necessary?

☞ Suppose you want to assign the object reference **o** to a variable of the **Student** type using the following statement:

```
Student b = o;   // A compile error would occur.
```

☞ Why does the statement **Object o = new Student()** work and the statement **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 you can see that **o** is really a **Student** object, the compiler is not so clever to know it.

Why Casting Is Necessary?

- To tell the compiler that **o** is a **Student** object, use an **explicit casting**.
- The syntax is similar to the one used for casting among primitive data types.
- Enclose the target object type in parentheses and place it before the object to be cast, as follows:

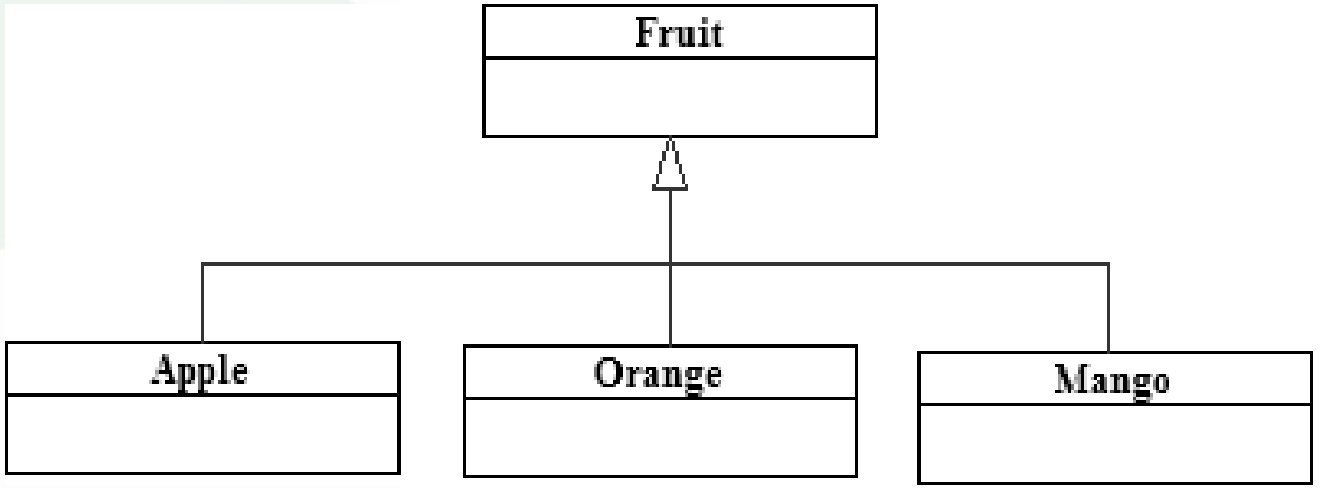
```
Student b = (Student) o ; // Explicit casting
```

Casting from Superclass to Subclass

- Explicit casting **must** be used when casting an object from a superclass to a subclass.

```
Fruit fruit = new Apple();
Apple a = (Apple) fruit;
Orange o = (Orange) fruit;
```

- This type of casting **may not** always succeed.



The `instanceof` Operator

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

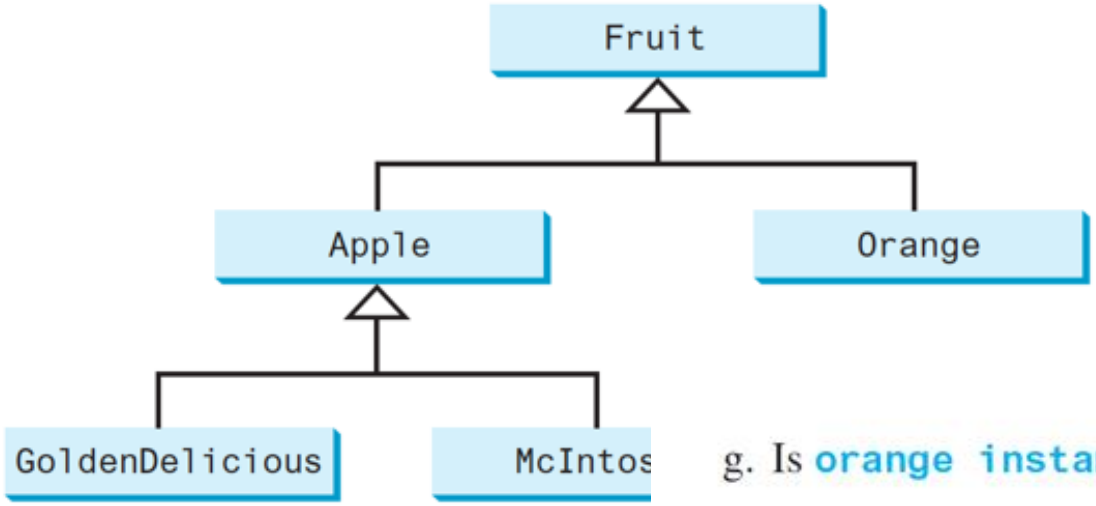
```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */

if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

Parentheses

Caution

The object member access operator (.) precedes the casting operator. Use parentheses to ensure that casting is done before the . operator, as in

```
((Circle)object).getArea();
```



Assume the following code is given:

```
Fruit fruit = new GoldenDelicious
Orange orange = new Orange();
```

Answer the following questions:

a. Is `fruit instanceof Fruit`?

b. Is `fruit instanceof Orange`?

c. Is `fruit instanceof Apple`?

d. Is `fruit instanceof GoldenDelicious`?

e. Is `fruit instanceof McIntosh`?

f. Is `orange instanceof Orange`?

g. Is `orange instanceof Fruit`?

h. Is `orange instanceof Apple`?

i. Suppose the method `makeAppleCider` is defined in `Fruit` invoke this method? Can `orange` invoke this

j. Suppose the method `makeOrangeJuice` is defined `orange` invoke this method? Can `Fruit` invoke this

k. Is the statement `Orange p = new Apple()` legal?

l. Is the statement `McIntosh p = new Apple()` leg

m. Is the statement `Apple p = new McIntosh()` leg

What's Wrong?

```
public class Test {
  public static void main(String[] args) {
    Object fruit = new Fruit();
    Object apple = (Apple)fruit;
  }
}

class Apple extends Fruit {
}

class Fruit {
}
```

The `equals` Method

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

```
public boolean equals(Object obj) {
  return this == obj;
}
```

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

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

NOTE

The == comparison operator is used for comparing two primitive data type values

or

For determining whether two objects have the same references.

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

The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element `o` at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element `o` at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element `o`.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element `o` from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : boolean`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

Generic Type

ArrayList is known as a generic class with a generic type E. You can specify a concrete type to replace E when creating an ArrayList. For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();

ArrayList<String> cities = new ArrayList<>();
```

TestArrayList Run

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

DistinctNumbers Run

Array Lists from/to Arrays

Creating an ArrayList from an array of objects:

```
String[] array = {"red", "green", "blue"};
  ArrayList<String> list = new
ArrayList<>(Arrays.asList(array));
```

Creating an array of objects from an ArrayList:

```
  String[] array1 = new String[list.size()];
  list.toArray(array1);
```

Sort, max and min in an Array List

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.max(
   new ArrayList<String>(Arrays.asList(array))));
java.util.Collections.sort(new
ArrayList<String>(Arrays.asList(array)));

String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.min(
  new ArrayList<String>(Arrays.asList(array))));
```

Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new
    ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

The MyStack Classes

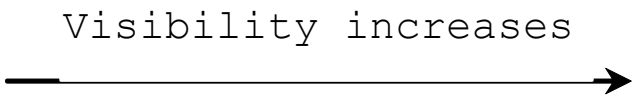
A stack to hold objects.

MyStack

MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public

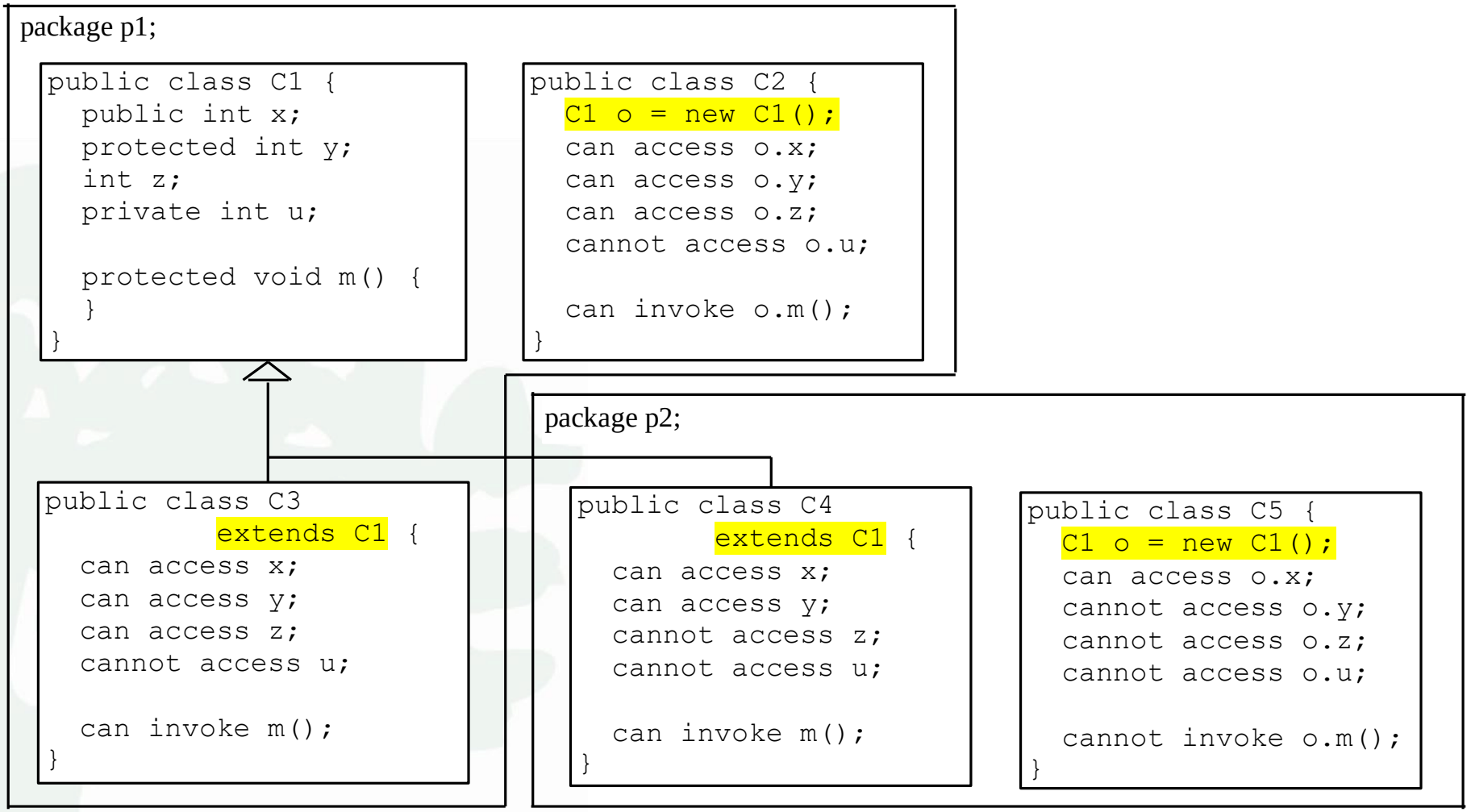


`private, none (if no modifier is used), protected, public`

Accessibility 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



A Subclass Cannot Reduce the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot Reduce the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.