Chapter 11
Inheritance and Polymorphism

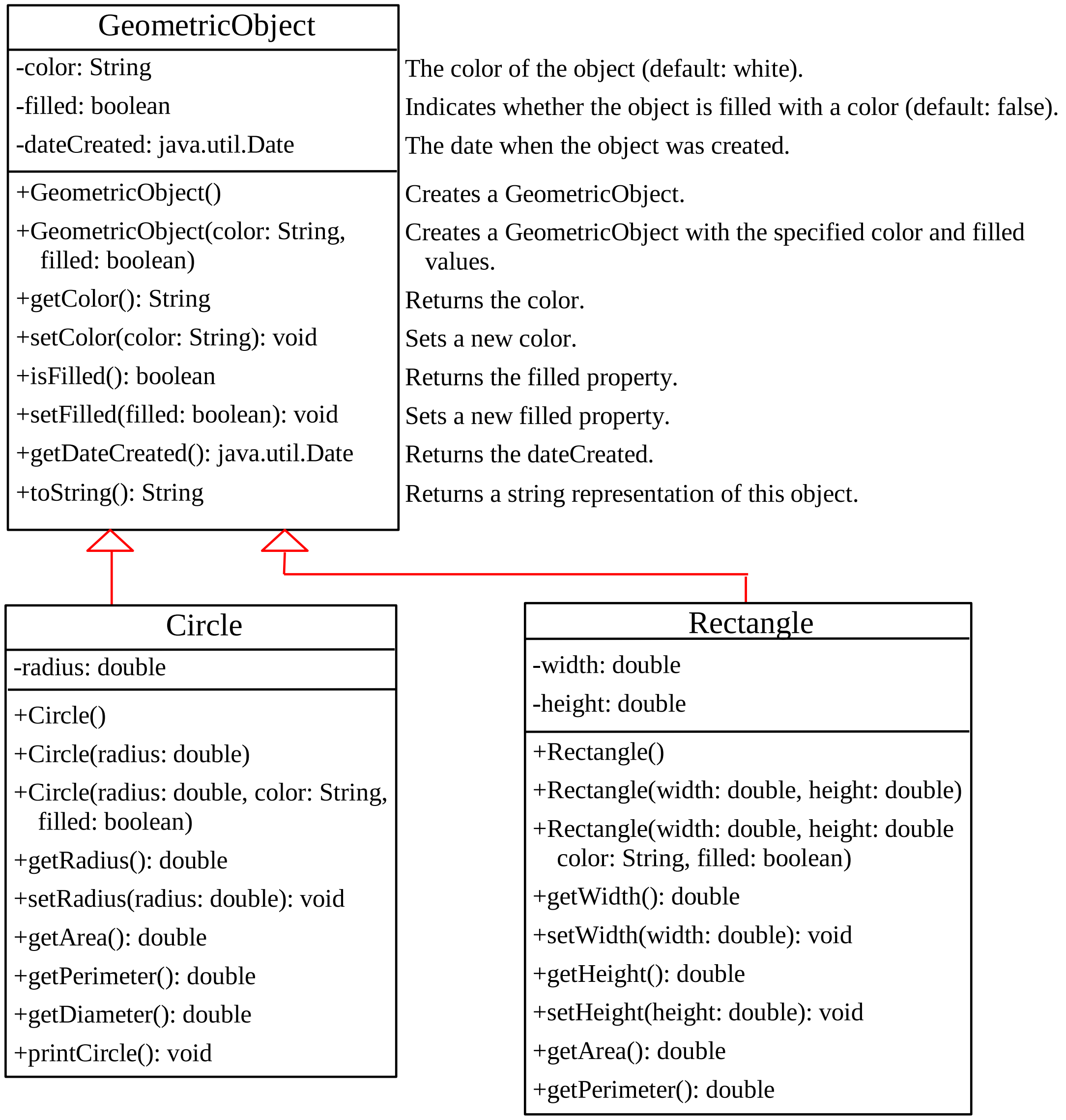
Motivation

OOP is built on three principles: Abstraction and Encapsulation (classes/objects, discussed in chapters 9 and 10), Inheritance, and Polymorphism.

Suppose you need to 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*. Inheritance allows defining subclasses from superclasses, where the subclass is more specialized than the superclass.

Can objects of a subclass be used anywhere objects of the superclass are expected? *The answer is to use polymorphism*. Polymorphism means that an object of the subclass can be used anywhere we expect an object of the superclass because an object of the subclass is an object of the superclass, but not vice versa.

Superclasses and Subclasses



GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Notes

- The subclass (child class) is NOT a subset of the superclass (parent class), it may include more attributes and methods due to inheritance.
- Private data fields are NOT accessible to the subclass. The subclass cannot directly access them even it inherits them!
- The superclass of all Java classes is the java.lang.Object class.
- Inheritance syntax is `extends:`

```
Public class Student extends Person
```

That is, class Student inherits everything in class Person and may add its own features (variables and methods).

Using the Keyword `super`

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

- To call a superclass constructor method
- To call a superclass *public* or *protected* method

Notice that *protected methods* are accessible by subclasses only.

Are Superclass's Constructors Inherited? NO

A constructor is used to construct an instance (object) of the class. Unlike data fields 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`***. to explicitly construct the parent's part of a child class object.

Note: *If the keyword* ***super*** *is not explicitly used, the superclass's no-arg constructor is automatically invoked* by default.

With inheritance, an object of a subclass is constructed by invoking the constructors of all superclasses along the inheritance path.

In other words, each superclass constructor builds that superclass part of the subclass object.

CAUTION

Invoking the parent class constructor in a child class causes a syntax error. You must use the keyword super to call the parent class constructor. Java requires that the call statement appears first in the child class constructor.

```
public Child(int a) {
 //some statements
}
```

is equivalent to

```
public Child(int a) {
 super(x,y); //call
 //some statements
}
```

Example

A constructor may invoke an overloaded constructor (in the same class) or in the parent class. If none of them is invoked explicitly, the compiler adds super() as the first statement in the constructor method. For example,

```
public Child() {
}
```

is equivalent to

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

```
public Child(int d) {
   //some statements
}
```

is equivalent to

```
public Child(int d) {
   super();
   //some statements
}
```

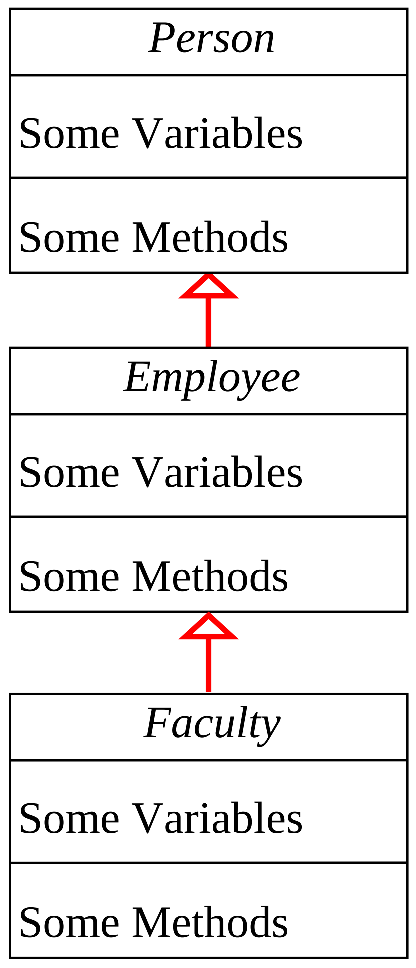
Constructor Chaining

Constructing an instance of a class invokes all parent classes constructors along the inheritance path. This is known as *constructor chaining.*

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

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


Person
Some Variables
Some Methods
Employee
Some Variables
Some Methods
Faculty
Some Variables
Some Methods

animation

Trace Execution

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

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

1. Start from the main method

animation

Trace Execution

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

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

2. Invoke Faculty constructor

animation

Trace Execution

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

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

3. Invoke Employee’s no-arg constructor

animation

Trace Execution

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

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

4. Invoke Employee(String) constructor

animation

Trace Execution

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

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

5. Invoke Person() constructor

animation

Trace Execution

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

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

6. Execute println

animation

Trace Execution

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

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

7. Execute println

animation

Trace Execution

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

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

8. Execute println

animation

Trace Execution

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

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

9. Execute println

Output

```
----jGRASP exec: java -ea Faculty
(1) Person's no-arg constructor is invoked
(2) Invoke Employee’s overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked

----jGRASP: operation complete.
```

Defining a Subclass

A subclass inherits from a superclass. You can also:

- Add new properties (data fields)
- Add new methods
- Override methods of the superclass

Calling Superclass Methods

It is possible to explicitly call a method in the parent class using the keyword **`super`**.

For example, you could rewrite method **`printCircle()`** in class Circle as follows:

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

Overriding Methods of 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 a superclass. This is referred to as *method overriding.*

```
public class Circle extends GeometricObject
{ //Other methods are omitted
  //Override method toString defined in GeometricObject

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

}
```

NOTES

- An instance method (i.e., non-static) can be overridden only if it is accessible (public or protected). Thus, a private method cannot be overridden, because it is not accessible outside its own class.

- If a method defined in a subclass is also defined as private in the superclass, then the two methods are completely unrelated due to their limited visibility.

- 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, then the method defined in the superclass is hidden from the subclass.

Overriding vs. Overloading

```java
public class Test1 {
  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);
  }
}
```

```
----jGRASP exec: java -ea Test1
10.0
10.0
----jGRASP: operation complete.
```

```java
public class Test2 {
  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);
  }
}
```

```
----jGRASP exec: java -ea Test2
10
20.0
----jGRASP: operation complete.
```

The Object Class and its Methods

Every class in Java is a child of class java.lang.Object.

If no inheritance is specified when class A is defined, the superclass of class A is by default class Object.

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

Equivalent

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

Method toString()

Method toString() returns a string representation of the object. The default implementation returns a string consisting of a class name, the at sign (@), and a number representing the object's memory address.

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

The code displays something like `Loan@160a7e5`.

This message is not very helpful or informative. Always override method toString() so that it returns a meaningful string representation of the object.

Method toString()

```
public String toString(){
//amount and years are instance variables in class Loan
return ("The loan is " + amount + " dollars for " + years + " years.");
}
```

```
Loan myLoan = new Loan(5000, 10); //new loan object
System.out.println(myLoan.toString());//print object
System.out.println(myLoan);//print object
```

<u>Output:</u>

```
    The loan is 5000 dollars for 10 years.

    The loan is 5000 dollars for 10 years.
```

Polymorphism

Polymorphism means that an object of the subclass can be used 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 means that a variable (reference) of a supertype can refer (point) 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**.

Think of classes Apple and Orange that extends class Fruit.

Polymorphism

Assume that classes Apple and Orange extend class Fruit.

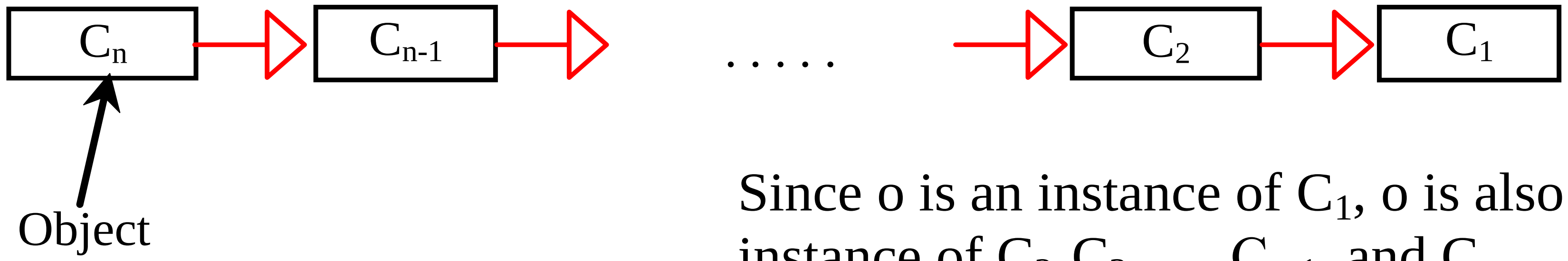
```
public class Apple extends Fruit {...}
public class Orange extends Fruit {...}
...
public class testFruits {
  public static void main (String[] args){
  Fruit F1, F2;  Apple A;  Orange O;
  ...       // create the objects
  F1 = A; // safe assignment
  F2 = O; // safe assignment
  A = F1; // unsafe assignment, must use casting
  O = F2; // unsafe assignment, must use casting
  A = (Apple)F1;  // now it is safe assignment
  O = (Orange)F2; // now it is safe assignment
 ...
}
```

Dynamic Binding

Dynamic binding 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 method p, the JVM searches for the implementation for 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/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 (at compile time) a method signature and binding (at runtime) a method implementation 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 Java Virtual Machine dynamically binds the implementation of the method to a call 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 (which is the parent of all Java classes). 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(), the implementation to use will be determined dynamically at runtime by the Java Virtual Machine. This capability is known as *dynamic binding*.

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 of a superclass type (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 in the object being invoked (e.g., toString()) is determined dynamically at runtime.

Output

```
----jGRASP exec: java -ea PolymorphismDemo
Student
Student
Person
java.lang.Object@15db9742

----jGRASP: operation complete.
```

Implicit Casting of Objects

We have already used the casting operator to convert variables of one primitive type to another. *Casting* can also 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 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. (i.e., an object of the child class is an object of the parent class)

Explicit Casting of Objects

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

```
    Student b = o; //compile error!!
```

Question: Why does `Object o = new Student();` work fine but

```
        Student b = o;
```

doesn't?

Answer: 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 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
```

TIP

To help understand casting, you may also re-consider the analogy of fruit, apple, orange, banana, etc.. with the Fruit class as the superclass for classes Apple, Orange, Banana, etc…

An apple is a fruit, so you can always safely assign an instance of Apple to a variable of type Fruit.

```
Fruit fruit;
Apple apple = new Apple();
fruit = apple;
```

However, a fruit is not necessarily always an apple, so you have to use explicit casting to assign an instance of class Fruit to a variable of type Apple, Orange, or Banana as shown below.

```
Fruit fruit = new Fruit();
Apple  x = (Apple)fruit;
...
Orange y = (Orange)fruit;
...
Banana z = (Banana)fruit;
...
```

The **`instanceof`** Operator

Use the operator **`instanceof`** to test/check whether an object is an instance of a given 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());
  System.out.println("The circle area is " +
                    ((Circle)myObject).getArea());
  ... // Some lines of code
}
... // Some lines of code
```

Method **`equals()`** in Class Object

Method **`equals()`** in class Object is used to compares the **content** of two objects. The default implementation of this method in the Object class is as follows:

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

For example, overriding method equals() 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/check whether two objects have the same content, 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 the two reference variables refer to the same object in the memory.

Class ArrayList

Arrays are static (fixed size). Array lists are dynamic. Java provides class ArrayList that can be used to store 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 object.

Arraylists store ONLY objects!

Examples:

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

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

    ArrayList<java.util.Date> dates = new ArrayList<>();

    ArrayList<Integer> grades = new ArrayList<>();

    ArrayList<Double> measurements = new ArrayList<>();

    ArrayList<Circle> myCircle = new ArrayList<>();

    ArrayList<Fruit> myFruits = new ArrayList<>();
```

Arrays vs. 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();`

Useful Methods:
ArrayLists from/to Arrays

Creating an ArrayList from an array of objects (not primitive type):

```
String[] myColors = {"red", "green", "blue"};
ArrayList<String> colorsList = new ArrayList<>(Arrays.asList(myColors));
// method asList() is defined in class Arrays
```

Creating an array of objects from an ArrayList:

```
String[] myArray = new String[myList.size()]; //myList is arraylist

myList.toArray(myArray);
// method toArray() is defined in class Arrays
```

Useful Methods: min, max, and shuffle on ArrayLists

```
String[] colors = {"red", "green", "blue"};
ArrayList<String> list = newArrayList<>(Arrays.asList(colors));
System.out.pritnln(java.util.Collections.max(list));  <==
System.out.pritnln(java.util.Collections.min(list));  <==
//methods min() and max() is defined in class Collections

Integer[] nums = {83, 51, 98, 14, 15, 34, 31, 26, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(nums));
java.util.Collections.sort(list); <==
System.out.println(list);
java.util.Collections.shuffle(list); <==
System.out.println(list);
//methods sort() and shuffle() are defined in class Collections
```

The **`protected`** Modifier

- The **`protected`** modifier can be applied on data and methods in a class. A protected variable or 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

`Visibility increases`

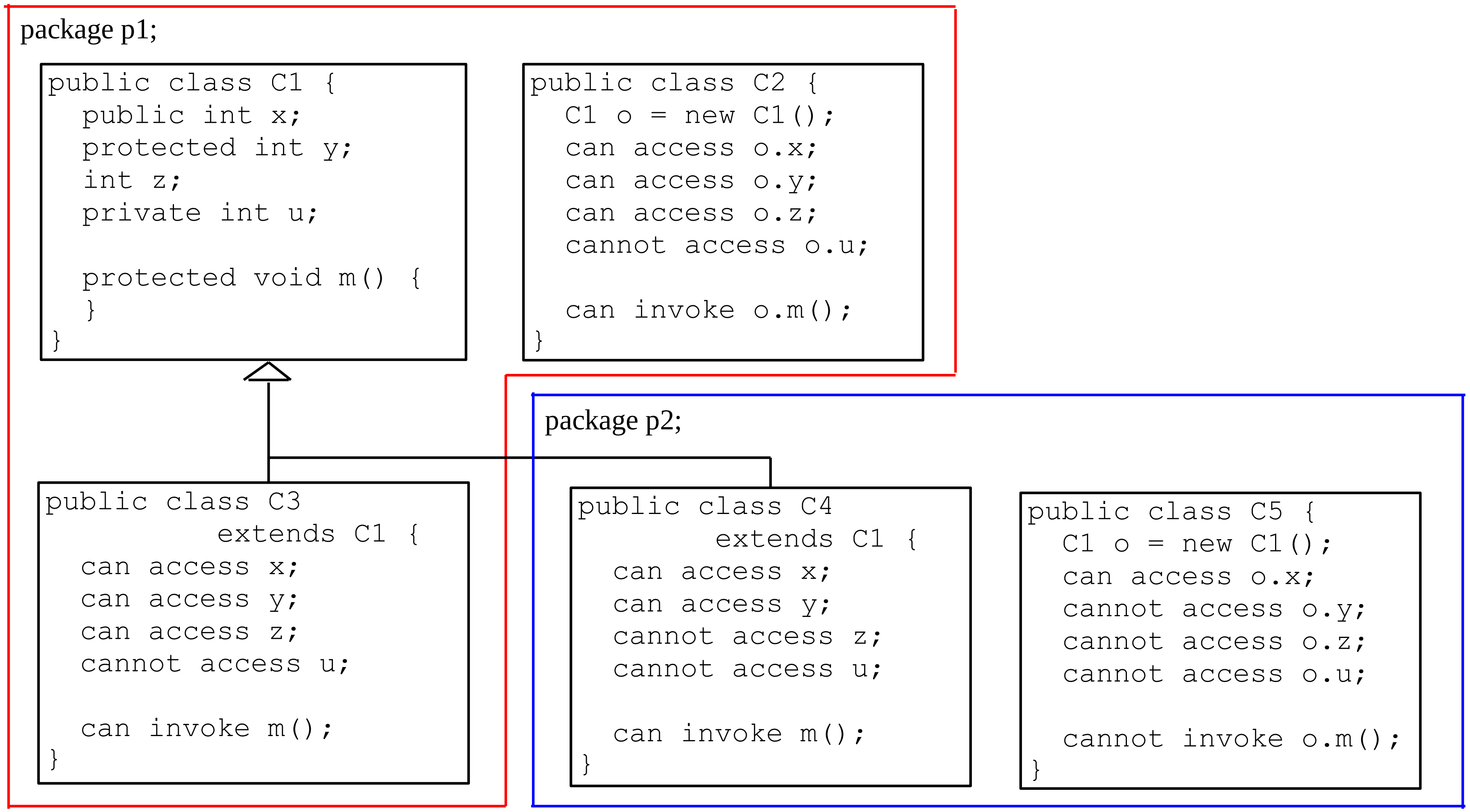
$\longrightarrow$

`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



```
package p1;

public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

```
package p2;

public class C4
          extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

A Subclass Cannot Weaken Accessibility in the Superclass

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken (restrict) 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.

`Visibility increases`

$\xrightarrow{\hspace{6cm}}$

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

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.

End of Chapter