CS115
Principles of Computer Science

Chapter 9 Inheritance , Polymorphism, and ArrayList

Prof. Joe X. Zhou
Department of Computer Science

Re: Objectives in Strings and Text I/O

- **To use the String class to process fixed strings**
- **To use the Character class to process a single character**
- **To use the StringBuilder/StringBuffer class to process flexible strings**
- **To know the differences between the String, StringBuilder, and StringBuffer classes**
- **To learn to pass strings to the main method from the command line**
- **To discover file properties, delete and rename files using the File class**
- **To write data to a file using the PrintWriter class**
- **To read data from a file using the Scanner class**

Objectives in Inheritance and Polymorphism

- To develop a subclass from a superclass through *inheritance*
- To invoke the superclass's constructors and methods using the super keyword
- To override methods in the subclass
- To distinguish differences between overriding and overloading
- To comprehend polymorphism, dynamic binding, and generic programming
- To describe casting and explain why explicit downcasting is necessary
- To store, retrieve, and manipulate objects in an `ArrayList`
- To restrict access to data and methods using the *protected* visibility modifier
- To declare constants, unmodifiable methods, and nonextendable classes using the *final* modifier

Inheritance

- What is inheritance? Why we need it?
- Inheritance allows us to derive new classes from existing classes!
- A class C1 inherits from another class C2

```
public class C1 extends C2 {
……
}
```

  - C2: superclass / parent class
  - C1: subclass / child class
  - C1 inherits *accessible* data fields and methods from C2, and may also add new data fields and methods.
  - No multiple inheritance in Java

- Inheritance is to model the *`is-a`* relationship!
  - Circle is a geometric object
  - Rectangle is a geometric object
  - Rectangle is NOT a circle

Declaring a Subclass

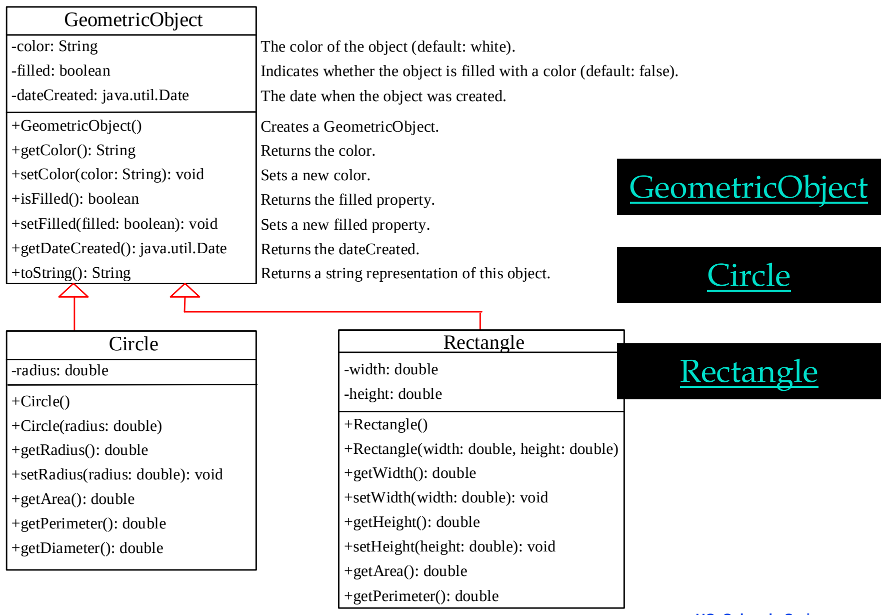
°A subclass extends properties and methods from the superclass.

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

What cannot be inherited?

constructors, private data fields and methods

Superclasses and Subclasses


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+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.
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.
GeometricObject
Circle
Rectangle
Circle
-radius: double
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Example: TestCircleRectangle

```
package chapter9;

public class TestCircleRectangle {
  public static void main(String[] args) {

    Circle circle = new Circle(1);
    System.out.println("A circle " + circle.toString());
    System.out.println(circle.getRadius());
    System.out.println("The radius is " + circle.getRadius());
    System.out.println("The area is " + circle.getArea());
    System.out.println("The diameter is " + circle.getDiameter());

    Rectangle rectangle = new Rectangle(2, 4);
    System.out.println("\n " );
    System.out.println(" A rectanlge " + rectangle.toString());
    System.out.println("The area is " + rectangle.getArea());
    System.out.println("The perimeter is " + rectangle.getPerimeter());
  }
}
```

Are superclass's Constructor Inherited?

- What can not be inherited?
  - Private data fields and methods
  - constructors
- Can a subclass invoke a superclass's constructor(s)?
  - They are invoked explicitly or implicitly.
  - Explicitly using the *super* keyword.
  - Implicitly: if the keyword *super* is not explicitly used, the superclass's no-arg constructor is automatically invoked
    - so, if a class is to be extended, better provide a no-arg constructor

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
}
```

Constructor Chaining

•Constructor chaining: constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain(by default, all no-arg constructors).

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }
  public Faculty() {
    System.out.println(" Faculty's no-arg constructor is invoked");
  }
}

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

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

Trace Code

Impact of a Superclass without no-arg Constructor

Principle: a class must have some constructor, or use its superclass' constructor either explicitly or implicitly.

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");
  }
}
```

If a class is to be extended, you better to provide no-arg constructor!

Examples of Constructor Chaining

What is the printout of running the class, or any problem?

```
class A {
   public A() {
      System.out.prinln("A constructor: Hello");
   }
}

class B extends A {
}

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

Examples of Constructor Chaining

What is the printout of running the class, or any problem?

```
class A {
   public A(int x) {
      System.out.prinln(x);
   }
}

class B extends A {
}

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

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: super.method(parameters);**

```
public void printCircle() {
  System.out.println(“The circle is created “ + super.getDateCreated() );
}
// but a chain of supers is illegal in Java
```

But when need to call a super.method(), since it must be inherited!

Re: Declaring a Subclass

°A subclass extends properties and methods from the superclass.

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

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;
  }
}
```

Q: after overriding, can an instance of Circle invoke toString method defined in the superclass GeometricObject class? Not anymore!

NOTE

•An instance method can be overridden only if it is accessible. Can a private method be overriden?

- 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.

• 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.

Q: What is overloading? Can the overloading be done in a class hierarchy? What is the difference between overloading and overriding?

Overriding vs. Overloading

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

class B {
  public void p(int i) {
  }
}

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

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

class B {
  public void p(int i) {
  }
}

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

• **Overriding: same signature and same return type!**

• **Overloading: same name, but with different signatures to distinguish them!**

Q: what are output of the two Test classes above, respectively?

The `Object` Class

° Every class in Java is inherited 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 {
  ...
}
```

$\Updownarrow$

```
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.

```
Loan loan = new Loan();

System.out.println(loan.toString());
```

The code displays something like Loan@15037e5 . This message is not very helpful or informative.

Overriding in GeometricObject.java:

```
public String toString{
    return "created on " + dataCreated + "\ncolor: " + color
           + " and filled " + filled;
    }
```

So, toString method is often to be overridden in the subclasses!

Polymorphism / 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"; //method overriding
  }
}

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

Method *m* takes a parameter of the Object type. You can invoke it with any object (an instance of a subclass must be an instance of its superclass)

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

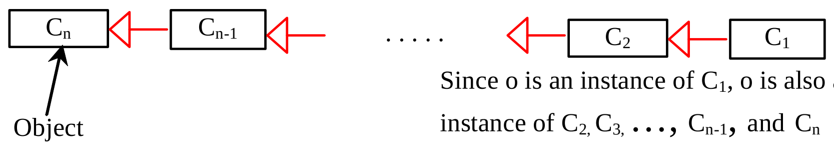
When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined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

Polymorphism Demo

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. Dynamic Binding

° **What is the key difference between matching a method in overloading and dynamic binding?**

° **Matching a method signature and binding a method implementation are two issues.**

° **Method matching deals with method *overloading* (same name, but different signatures): the compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.**

° **Dynamic binding associated with method *overriding*: a method may be implemented in several subclasses (with same signature and return type). The JVM dynamically binds the implementation of the method at runtime.**

So, what is the key benefit of 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 {
  public String toString(){
     return "Graduate Student";
  }
}

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

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

° **Suppose we have put four classes in four files. A programmer changes the GraduateStudent.java implementation as:**

```
class GraduateStudent extends Student {
    public String toString() {
        return "Graduate Student";
    }
}
```

° **Now, we have a new version of GradateStudent.java with a new toString method, do we need to recompile all four classes, or just GraduateStudent class?**

° **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). When an object (e.g., a Student object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically at runtime.**

Example: Demonstrating Polymorphism and Casting

What we have learned from Polymorphism?

let a method's parameter type be as "super" (say Object) as possible!

This example creates two geometric objects: a circle, and a rectangle, invokes one displayObject method to display the objects. It displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

Note that you are asked to use one displayObject method only!

Casting Objects

° **You 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. In the preceding section, the statement**

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

assigns the object of Student class (subclass) to a parameter of the Object class (superclass). This is *upcasting*. The statement is equivalent to:

```
Object o = new Student();       // Implicit casting for upcasting
m(o);                           // method m() takes object as parameter
```

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

Why Explicit Casting Is Necessary for Downcasting?

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

```
Student b = o;
```

• A compilation error would occur. This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

• Downcasting: cast an object of a superclass to a variable of its subclass. 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
```

What if o is Not a Student object indeed? Compile error? Runtime error?

Downcasting: from Superclass to Subclass

° Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may NOT always succeed.

° Example: consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

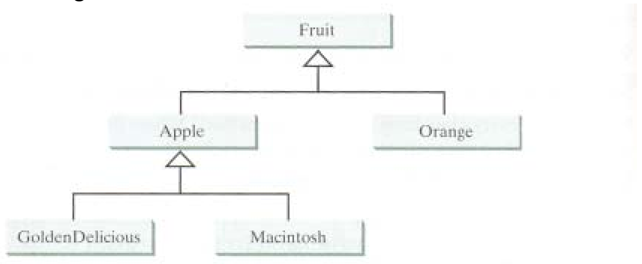
```
Fruit fruit = new Orange();
…..
Apple x = (Apple) fruit;
Orange y = (Orange) fruit;
```

What if fruit is an orange, but NOT an object of Apple class?
A runtime error ClassCastException occurs for the first casting.

How to ensure an object is an instance of a subclass before casting?
The *`instanceof`* operator!

The `instanceof` Example

Assume that *fruit* is an instance of GoldenDelicious and *orange* is an instance of Orange.



9.11 GoldenDelicious and Macintosh are subclasses of Apple, Apple and e are subclasses of Fruit.

(1) Is fruit instanceof Orange?
(2) Is fruit instanceof Apple?
(3) Is fruit instanceof GoldenDelicious?
(4) Is fruit instanceof Macintosh?
(5) Is orange instanceof Orange?
(6) Is orange instanceof Fruit?
(7) Is orange instanceof Apple?

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());
  ...
}
```

`Why Casting is necessary?`

The declared type decides which method to match at compile time.

`Why not declare myObject as a Circle type in the first place?`

To support polymorphism and generic programming, it is good to declare a variable with a superclass type which can accept a value of any subclass.

Example: Demonstrating Polymorphism and Casting

```
package chapter9;
                                                        * Object can be replaced by GeometricObject
public class TestPolymorphismCasting {
  public static void main(String[] args) {
    // Declare and initialize two objects
    Object object1 = new Circle(1);                    // implicit casting
    Object object2 = new Rectangle(1, 1);              // implicit casting

    displayObject(object1); // Display circle and rectangle
    displayObject(object2);
  }

public static void displayObject(Object object) {/** A generic method for displaying an object */
    if (object instanceof Circle) {                               // to ensure downcasting can be applied
       System.out.println("The circle area is " + ((Circle)object).getArea());       // explicit casting
       System.out.println("The circle diameter is " +  ((Circle)object).getDiameter());
    }
    else if (object instanceof Rectangle) {
       System.out.println("The rectangle area is " +  ((Rectangle)object).getArea());
    }
  }
}
```

TestPolymorphismCasting

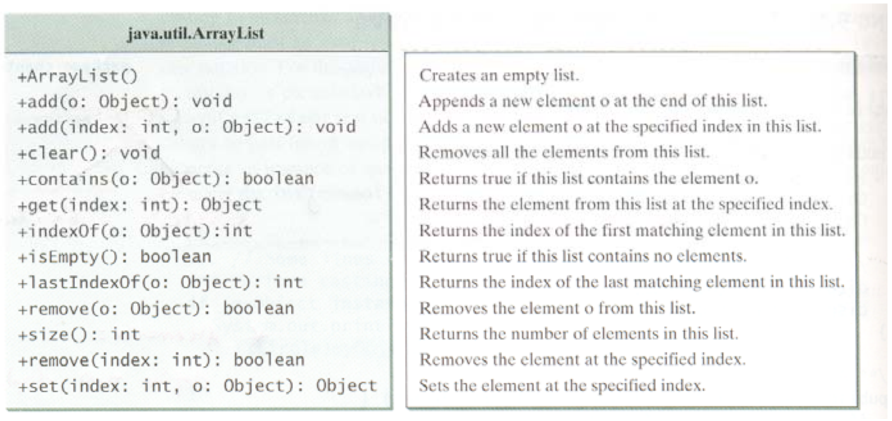
The displayObject method be invoked by passing any instance of Object, an example of generic programming!

Array vs. The *ArrayList* Class

° **What if we want to have a scalable array, instead of fixed-size array?**

° **Java provides `ArrayList` to store an unlimited number of objects.**

° **`ArrayList` methods:**



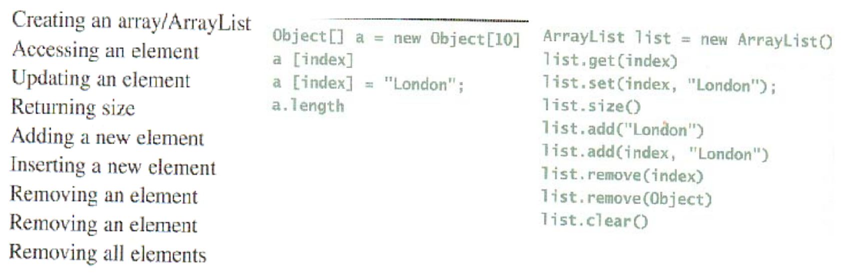
java.util.ArrayList	
+ArrayList()	Creates an empty list.
+add(o: Object): void	Appends a new element o at the end of this list.
+add(index: int, o: Object): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): Object	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: Object): Object	Sets the element at the specified index.

The *ArrayList* Example

```
public class TestArrayList {
  public static void main(String[] args) {
     java.util.ArrayList cityList = new java.util.ArrayList();

    cityList.add("London");
    cityList.add("New York");
    cityList.add("Paris");
    cityList.add("Toronto");
    cityList.add("Hong Kong");
    cityList.add("Singapore");
    System.out.println("List size? " + cityList.size());
    System.out.println("Is Toronto in the list? " +  cityList.contains("Toronto"));
    System.out.println("The location of New York in the list? " + cityList.indexOf("New York"));
    System.out.println("Is the list empty? " + cityList.isEmpty()); // Print false

    cityList.add(2, "Beijing");                       //arraylist index starts at 0 too.
    cityList.remove("Toronto");
    cityList.remove(1);
    String headCity = (String) list.get(0);    // explicit cast is necessary; maybe instanceof can be used as well.
    for (int i = 0; i < cityList.size(); i++)
       System.out.print(cityList.get(i) + " \n");

    // Create a list to store two circles
    java.util.ArrayList list = new java.util.ArrayList();
    list.add(new Circle(2));
    list.add(new Circle(3));

    // Display the area of the first circle in the list
    System.out.println("The area of the circle? " + ((Circle)list.get(0)).getArea());    //what if no explicit cast here?
  }
}
```

Difference & Similarity between `Array` and *ArrayList*

° **It is easy to add, insert, and remove elements in a list.**



Creating an array/ArrayList	`Object[] a = new Object[10]`	`ArrayList 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()`

Iteration in *ArrayList*

° **The query iterator() returns an object of type *Iterator*, which allows one to traverse through all of the elements of an ArrayList.**

```
// assume myList is an constructed ArrayList list, and
// each element can respond to the toString() method so can be printed.

for (Iterator iter = myList.iterator(); iter.hasNext() ; ) {
        System.out.println(iter.next());
}

Can be replaced by

 for (int i = 0; i < myList.size(); i++) {
        System.out.println(myList.get(i));
}
```

Re: The *ArrayList* Example

```
public class TestArrayListIterator {
  public static void main(String[] args) {
     java.util.ArrayList cityList = new java.util.ArrayList();

    cityList.add("London");
    cityList.add("New York");
    cityList.add("Paris");
    cityList.add("Toronto");
    cityList.add("Hong Kong");
    cityList.add("Singapore");

     for (int i = 0; i < cityList.size(); i++)
      System.out.print(cityList.get(i) + " \n");

    for (Iterator iter = cityList.iterator(); iter.hasNext(); )
      System.out.print(iter.next() + "\n");
   }
}
```

TestArrayListIterator

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

Visibility increases

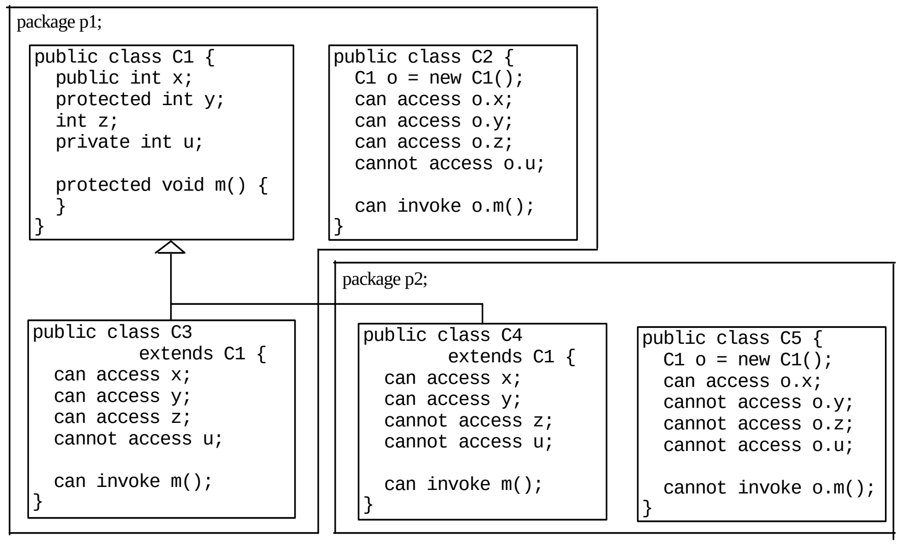
$\xrightarrow{\hspace{6em}}$

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



```
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();
}
```

```
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 the Accessibility

- Accessibility scaling is ok: a subclass may override a protected method in its superclass and change its visibility to public.
- Accessibility shrinking is not: a subclass cannot weaken 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.

The `final` Modifier

- What we have used final modifier for?
- The `final` variable is a constant:

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

- The `final` class cannot be extended:

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

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

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

- How we know if two variables of some primitive data type have the same value?
- How we know if the contents of two objects are the same?
- The equals() method compares the contents of two objects

```
String1.equals(string2);
Char1.equals(char2);
```

- The default implementation in the *`Object`* class compares whether *this* reference is equal to (“==”) the object reference passed, i.e., if the two refer to the same object. But it intends to be overridden since “==” is too strong!

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);
}
```

The == operator is stronger than the *equals* method, in that the == operator checks whether the two reference variables refer to the same object. 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; //cast
  }
  else
     return false;
}
```

Reading

- **Chapter 5 of the textbook: 5.12**
- **Chapter 9 of the textbook: 9.1 – 9.9, 9.11 – 9.12**
- **Do review questions**