Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Polymorphism

CS2012: Introduction to Programming II

Polymorphism

- four pillars of object oriented programming:
  - encapsulation
  - abstraction
  - inheritance
  - ***polymorphism***

- All classes define a type:
  - Circle defines the Circle type.
  - String defines the String type.

- ***subtype***: type defined by a subclass.
- ***supertype***: type defined by a superclass.

- Example: `Circle` is a ***subtype*** of `GeometricObject` and `GeometricObject` is a ***supertype*** of `Circle`

Polymorphism

- Inheritance enables a subclass to inherit features from its superclass with additional new features

- A subclass is a specialization of its superclass
  - ***every instance of a subclass is also an instance of its superclass, but the reverse is NOT true.***
  - Example: Every `Circle` is a `GeometricObject`, but not every `GeometricObject` is a `Circle`.

- You can always pass an instance of a subclass, to a parameter of its superclass type

- ***polymorphism***: a variable of a supertype can refer to a subtype object.

Dynamic Binding

Dynamic Binding

- ***dynamic binding***: a method can be implemented in several classes along the inheritance chain, and the JVM decides which method to invoke at runtime.

- As we have already seen, a method can be defined in a superclass and overridden in its subclass.
  - multiple versions of the same method exist along the inheritance chain.

Dynamic Binding

- Example: toString() is defined in the Object class, and overridden in GeometricObject

- Consider:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

- Which version of toString() gets called here?
  - The Object class's toString()?
  - The GeometricObject class's toString()?

Dynamic Binding

- a variable must declare a type

- lets introduce two new terms:
  - ***declared type***: the type that declares a variable
  - ***actual type***: the actual class for the object referenced by the variable

- In the previous example:
  - the declared type of o is `Object` because o is declared as `Object o =`
  - the actual type of o is `GeometricObject` because the variable o references an object created using `new GeometricObject()`

Dynamic Binding

- So, to answer the original question, Which version of `toString()` is called?
  - Answer: the version invoked is always determined by the actual type of the reference variable
- Therefore: the GeometricObject version is invoked.
- When does dynamic binding occur?: ***at runtime***
- The JVM will search along the inheritance chain, starting from the most subclass working towards the `Object` class until it finds a matching method.

Casting Objects and the `instanceof` Operator

Casting Objects and the `instanceof` Operator

- Remember casting?

```
char ch = 'k';
int n = (int)ch;
```

- Well we can ***typecast*** an object reference into another type of object reference.

- Two types of casting:
  - implicit
  - explicit

Casting Objects and the `instanceof` Operator

- ***implicit object casting***
  - assigning a reference of a subtype, to a reference of the subtype's supertype.
  - this is always allowed because a subtype is always an instance of its supertype....always.

- Example:
  - `GeometricObject go = new Circle();`

Casting Objects and the `instanceof` Operator

- explicit object casting
  - Example: is the following valid?

```
GeometricObject go = new Circle();
Circle c = go; //no this causes a compile error
```

- Why doesn't this work?
  - a `Circle` is always an instance of `GeometricObject`, but a `GeometricObject` is not necessarily an instance of `Circle`.

- Can we make this work? ….. Yes

```
GeometricObject go = new Circle();
Circle c = (Circle)go;
```

Casting Objects and the `instanceof` Operator

- For the casting to be successful, you MUST always make sure that the object to be cast is an instance of the subclass.

- If the superclass object is not an instance of the subclass a runtime `ClassCastException` occurs.

- Example:
  - If GeometricObject was not an instance of Circle, then it cannot be case into a Circle.

Casting Objects and the `instanceof` Operator

- How can you verify that a class is an instance of some other class?
  - the **`instanceof`** operator
    - NOTE: all lowercase
    - returns true if Class A is an **`instanceof`** Class B

- Example:

```
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());
  ...
}
```

Casting Objects and the `instanceof` Operator

- Why is casting objects even necessary?

- Consider this:
  - Circle is a subclass of `GeometricObject`
  - Circle defines a `getDiameter()` method
  - `GeometricObject` does not define a `getDiameter()` method.
  - Would the following code be valid?:

```
GeometricObject go = new Circle();
System.out.println(go.getDiameter());
```

Casting Objects and the `instanceof` Operator

- Other things to note:

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());
```

Casting a primitive type value is different from casting an object reference. Casting a primitive type value returns a new value. For example:

```
int age = 45;
byte newAge = (int)age; // A new value is assigned to newAge
```

However, casting an object reference does not create a new object. For example:

```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

Now reference variables `o` and `c` point to the same object.

The Object's equals method

The `Object's equals` method

- **`public boolean equals(Object o)`**
  - another method defined in the Object class.
  - tests whether two objects are equal:

- Example:
  - `object1.equals(object2)`

The `Object's equals` method

- The default implementation:

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

  - doesn't really do much for us
  - only returns whether or not both objects reference the same area in memory

- Good practice dictates that you should always override this method and implement your own for every class you create which could be compared to other instances of the same class.

The `Object`'s `equals` method

- Other Notes

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 overridden in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.

Caution

Using the signature `equals(SomeClassName obj)` (e.g., `equals(Circle c)`) to override the `equals` method in a subclass is a common mistake. You should use `equals(Object obj)`.

The `protected` Data and Methods

The `protected` Data and Methods

- Review:
  - ***private***: can be accessed only from inside of the class
  - ***public***: can be accessed from any other class

- What if you want to allow subclasses to access data fields or methods defined in the superclass directly, but prevent nonsubclasses from accessing these same data fields and methods?
  - Use the protected modifier

The `protected` Data and Methods

- Rules for Use:
  - `private`:
    - hide the members of a class completely so they cannot be accessed directly from outside the class.
    - use private for members which will not be used ANYWHERE outside the class
  - `no modifiers (default)`:
    - allow members of the class to be accessed directly from any class within the same package but not from other package.
  - `protected`:
    - enable the members of the class to be accessed by the subclasses in any package or classes in the same package
    - use protected if they are intended for extenders of the class, but not any other users of the class.
  - `public`:
    - enable members of the class to be accessed by any class.
    - use public if they are intended for users of the class

The `protected` Data and Methods

- Other Notes:
  - `private` and `protected` can only be used for members of the class
  - `public` and `default (no modifier)` can be used on members of the class as well as the class itself
  - a class with `no modifier` (a non public class) is not accessible by classes from other packages

Note

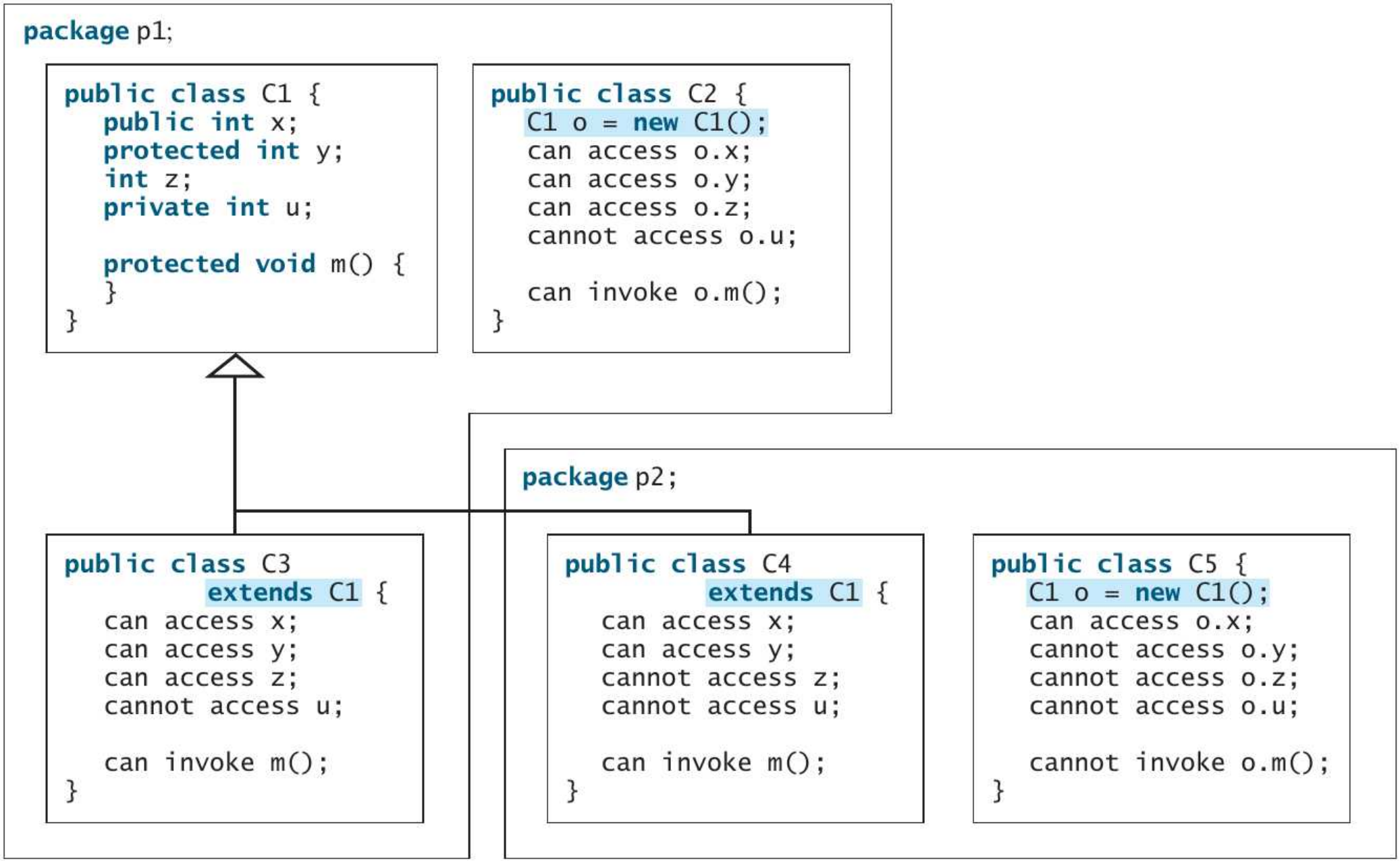
A subclass may override a protected method defined in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The `protected` Data and Methods

TABLE 11.2 Data and Methods Visibility

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

The protected Data and Methods



Preventing Extending and Overriding

Preventing Extending and Overriding

- A **final** class cannot be extended
- A **final** method cannot be overridden
- A **final** data field is a constant

- If you want to prevent a class from being extended (subclassed) or a method from being overridden by its subclasses you can use the `final` keyword.

- Examples: Math, String, StringBuilder, StringBuffer are all final, you cannot make subclasses from them

Preventing Extending and Overriding

```
public final class A {
  // Data fields, constructors, and methods omitted
}
```

```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```

References

- Liang, Chapter 11