INHERITANCE

Introduction

- Inheritance makes it possible to build new classes from existing classes, thus facilitating the reuse of methods and data from one class in another. Moreover, inheritance allows data of one type to be treated as data of a more general type.
- *Inheritance* is the mechanism that allows us to reuse the attributes and methods of one class in the implementation of another class.
- The child class can create its own attributes and methods as well as provide its own implementation of any method, new or inherited.

Access Modifiers

- A private variable or method is visible only to its defining class.
- A public variable or method is visible to any class.
- A protected variable or method is visible to its defining class and all its subclasses, as well as any other classes in the same package.

	Class	Package	subclass	World
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No

Subclasses

- A subclass inherits all public and protected methods of a base class unless the subclass overrides a method, thus providing its own implementation.
- A subclass does not inherit the constructors of the base class. The constructors of a base class are not considered constructors of a subclass.
- A derived class inherits the data and methods of the base class that are not private.
- Note, that a subclass may not override a public method with a private access modifier. In general, you may not assign more restrictive access privileges to an overridden method.

Super

- A child class may invoke a parent constructor using the keyword super.
- If super is used, then it must be the first statement of a constructor.
- We note that if a base class constructor is not explicitly called using super , the default constructor of the base class is automatically invoked. In this case, if the default constructor of the base class does not exist, a compilation error results.
- If a derived class overrides a method x() , the base class version of x() is still available to the derived class and can be invoked using the keyword super:
- super.x().

The IS-A Relationship

- The relationship between the base class and a derived class is termed an *is-a* relationship because every derived class *is-a* (kind of) superclass.
- When deciding whether or not to extend a class, you should determine whether or not an *is-a* relationship exists. If not, inheritance is probably inappropriate.

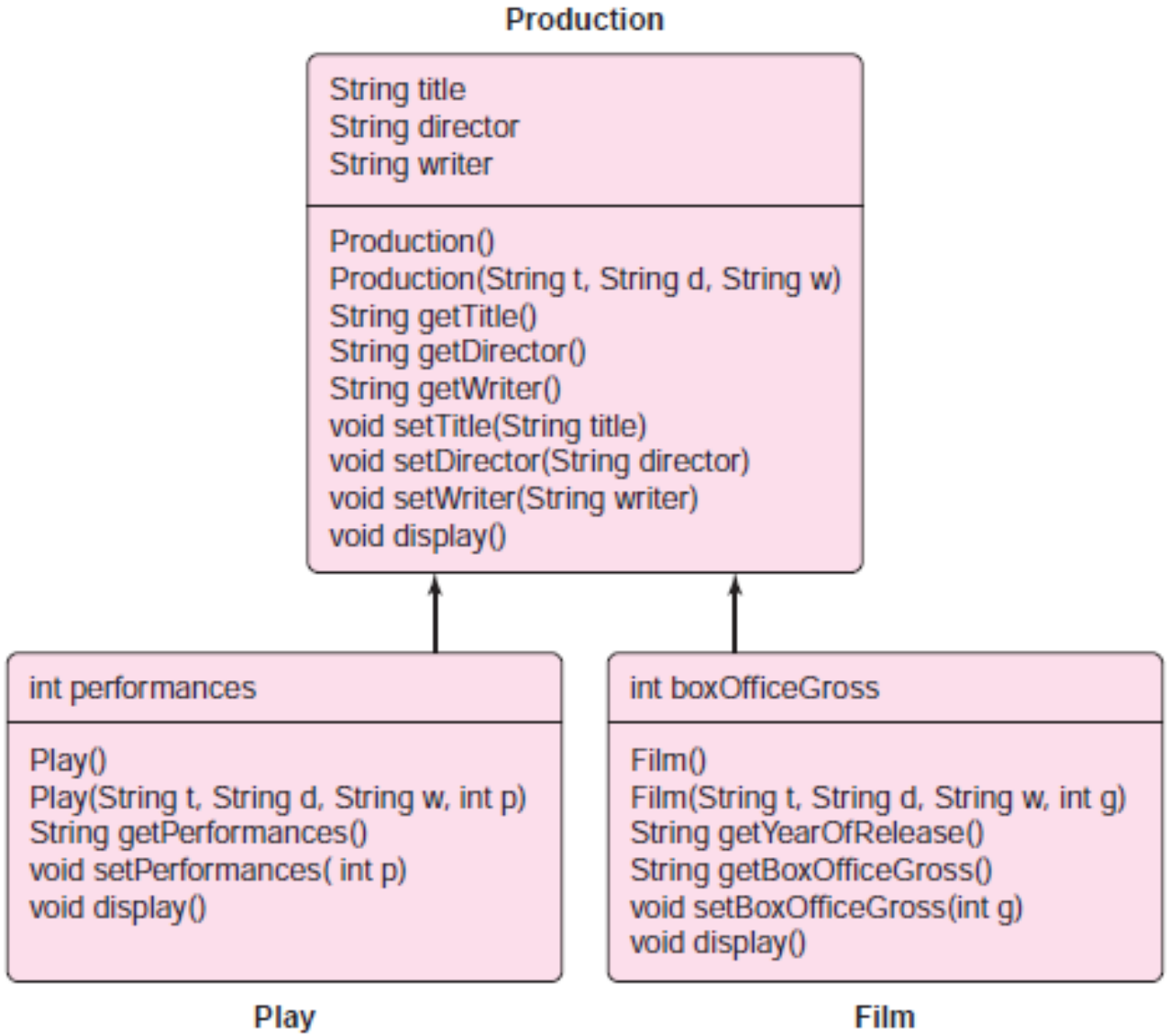

FIGURE 12.5 *Play* extends *Production*; *Film* extends *Production*

Inheritance VIA abstract classes

- An abstract class is a class that cannot be instantiated. However, an abstract class can be inherited.
- In general the abstract class has following properties
  - The keyword abstract denotes an abstract class. For example,
    - public **abstract** class Production

    specifies that Production is an abstract class.
  - An abstract class *cannot* be instantiated. You cannot create an object of an abstract class.
  - An abstract class can be inherited by other classes. Indeed, an abstract class is designed for inheritance, not instantiation.
  - An abstract class *may* contain abstract methods. An abstract method is a method with no implementation. For example, the method
    - public **abstract** void display() **;** // method has no body

    is an abstract method. Notice the keyword abstract and the terminal semicolon.

Continue..

- If an abstract class contains abstract methods, those methods *must* be overridden in any non-abstract subclass; otherwise the subclass is also abstract.
- All abstract classes and methods are public.
- To be of any use, an abstract class must be extended.
- It’s a kind of a contract.

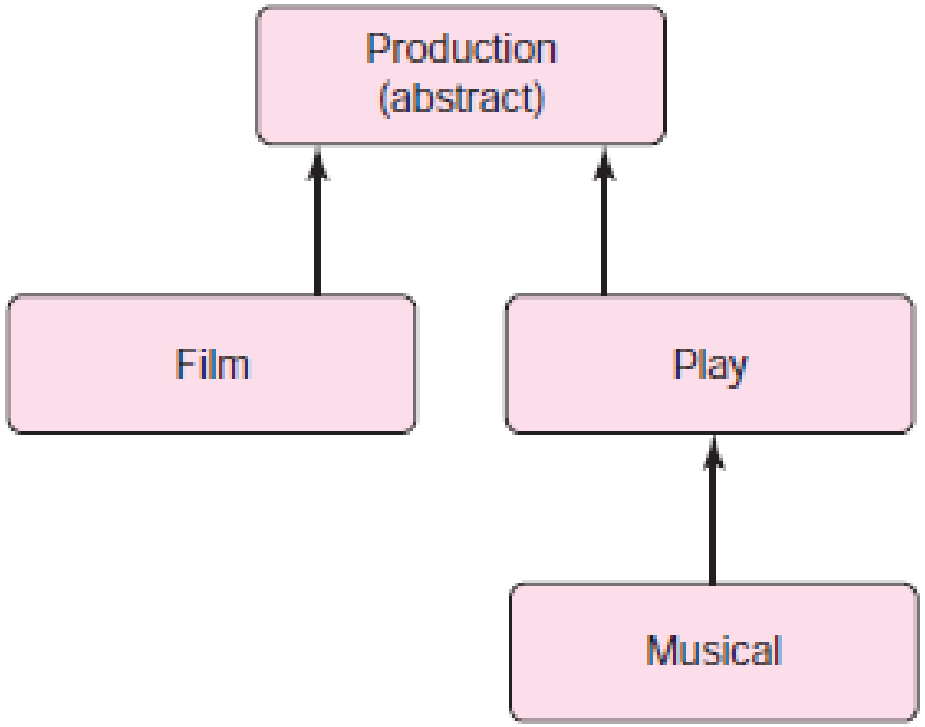

FIGURE 12.6 The *Production* hierarchy

Upcasting and Downcasting

- Upcasting is a language feature that allows a base-type reference to refer to an object of a derived type. Objects of a derived type may be considered objects of the base type.
    - Production p = new Film(),
    - Production q = new Play(), and
    - Production r = new Musical()
- Downcasting means casting an object to a derived or more specialized type. To invoke a derived class method using a base class reference, a downcast is necessary.
    - Play play = new Musical();
    - Musical musical = (Musical)play;
    - musical.getComposer();

Continue..

What would happen?

1. Play play = new Musical(…);
2. String name = play.getComposer() ;

line 1 is a valid upcast. However, the compiler complains about the method call on line 2. To the compiler, play is a Play reference and Play has no getComposer() method. So the compiler generates an error.

- Yet, because a Musical object is instantiated (line 1), an explicit downcast informs the compiler that play refers to a Musical object and fixes the problem:
  - Play play = new Musical(…);
  - String name = (**(Musical)**play).getComposer() ;

Continue..

- Casting does not change the actual object type. Only the reference type gets changed.
- Upcasting is always safe and never fails.
- Downcasting can risk throwing a ClassCastException, so the instanceof operator is used to check type before casting.

instanceof Operator

- Like && and ||, instanceof is a boolean operator that requires two operands.
    - *object* **instanceof** *class*
- If *object* belongs to or is derived from *class* , then instanceof returns true , otherwise instanceof returns false.
- For example,
    - Play play = new Play();
    - Musical musical = new Musical();
    - Film film = new Film();

Continue..

- Then,
    - film instanceof Film returns true,
    - film instanceof Production returns true,
    - musical instanceof Play returns true,
    - musical instanceof Film returns false, and
    - musical instanceof Production returns true.

- The instanceof operator can help a programmer to avoid casting errors.

Everything inherits: The Object Class

- The package java.lang , which is automatically imported into every application, contains Java's Object class. That's Object with an uppercase O.
- Every class is a subclass of Object. Every class is derived from Object.
- Every class extends Object. Math, String , and Scanner all extend Object. Play , Film , Musical ,Remote , and DirectRemote also extend Object. Film *is-an* Object; Play *is-an* Object . There is no escape; everything *is-an* Object . Object is the mother of all classes.
- Being a descendent of Object brings several familial privileges.

Continue..

- Every class inherits methods
    - public boolean equals(Object object) , and
    - public String toString()
  - from Object .
- Because every class extends Object , every class can be upcast to Object .
- For example,
    - Object remote = new Remote();
    - Object film = new Film();
- Overridding the equals(Object object) & toString() methods

Interfaces

- The English word *interface* can mean anything from the buttons on a TV to the public methods of a class. However, in Java, the term *interface* has a very specific meaning.
  - An *interface* is a named collection of static constants and abstract methods. An interface specifies certain actions or behaviors of a class but not their implementations.
- Unlike a class,
  - all methods of an interface are public ,
  - all methods of an interface are abstract , that is, there are no implementations at all,
  - an interface has no instance variables.
- Like an abstract class, an interface cannot be instantiated. In contrast to an abstract class, a class *does not extend* an interface. Instead, a class *implements* an interface.

Continue..

- An interface is a contract
  - A class that implements an interface must implement all the methods of the interface, or be tagged as abstract.
  - An interface provides a contract as well as a large dose of flexibility.
- Difference b/w interface and abstract class
  - As we have stated, an *is-a* relationship should hold between an abstract class and any subclass. However, the *is-a* relationship between a parent and child class need not hold between an interface and an implementing class.
  - There is not necessarily any commonality among classes that implement a particular interface other than a shared collection of methods that each class must implement. On the other hand, classes that extend a particular abstract class usually share some instance variables and method implementations.

Continue..

- Classes that extend the same abstract class share instance variables and perhaps also some code, but classes that implement the same interface do not necessarily have anything in common except a collection of methods that each class must implement.
- All **variables** declared inside **interface** are implicitly public static final **variables**(constants). All **methods** declared inside Java **Interfaces** are implicitly public and abstract, even if you don't use public or abstract keyword.

Multiple inheritance and interfaces

- Some object-oriented languages such as C++ allow *multiple inheritance.* Multiple inheritance means that a subclass can inherit from more than one base class. The unrestricted use of multiple inheritance is a controversial feature with many complexities and pitfalls.
  - For example, suppose that class A implements a display() method and class B implements a different display() method. If class C extends both A and B but does not override display() , which display() method does C inherit? There is no clear answer.
- Java Does not support multiple inheritance
  - A subclass cannot inherit from two different base classes.
  - A class may extend one class as well as implement any number of interfaces.

Upcasting to an interface

- The real power of an interface lies in upcasting.
- A derived class can be upcast to any one of its interfaces.
- For example, In particular, the Circle, Square , and Triangle objects of can be upcast to Geometry. So, for example, a single array can store *any* object that implements Geometry.

The Comparable interface

- As Java provides a plethora of ready-made classes, Java also provides a large number of ready-made interfaces. Among one of the most useful Java-supplied interfaces is the Comparable interface. Comparable is an interface with just one method, compareTo(...) :
    - public interface Comparable
        - {
            - int compareTo(Object o);
        - }
- In practice, compareTo(...) is *usually* implemented so that
    - a.CompareTo(b) −1 if a is less than b ,
    - a.CompareTo(b) 0 if a equals b , and
    - a.CompareTo(b) 1 if a is greater than b .
- A class that implements Comparable is advertising to its clients that its objects can be “compared.”

Composition and has a relationship

- Inheritance, as you know, is characterized by an *is-a* relationship:
  - a Square *is-a* Shape ,
  - a RightTriangle *is-a* Shape ,
  - a Film *is-a* Production ,
  - a Dog *is-an* Animal , and
  - a Bloodhound *is-a* Dog .

Continue..

- A person is *not* a BankAccount and a BankAccount is *not* a Person . There is no apparent or logical reason to consider a Person a type of BankAccount or vice versa. Inheritance is not a good fit.

```
public class Person
{
    private String name:
    private String address;
    // etc.
}
```

```
public class BankAccount
{
    private String accountNumber;
    private double balance;
    . . .
    public double balance()
    // etc.
}
```

Continue..

- Suppose, however, that every Person possesses a BankAccount . You have already seen that one object may contain objects of another class.
- In such a case, the relationship between the Person and the BankAccount classes is a *has-a* relationship. A Person *has-a* BankAccount . And a Person class can be defined with a BankAccount attribute.
- The relationship between Person and BankAccount is an example of *composition* —a relationship in which one object is composed of other objects.

```
public class Person
{
   private String name:
   private String address;
   private BankAccount account;
   // etc.
}
```

Continue..

- As an *is-a* relationship indicates inheritance, a *has-a* relationship signals composition.
- Inheritance implies an extension of functionality and the ability to upcast; composition indicates ownership. Inheritance and composition are very different concepts; the two should not be confused.