Java Programming

U Hou Lok

Department of Computer Science and Information Engineering,
National Taiwan University

Java 273
22 Aug.–2 Sep., 2016

First IS-A Relationship

- OO allows classes to inherit commonly used states and behaviors from other classes.
- This is called inheritance.
- So the classes exist in some hierarchy.
- A class can be declared as a subclasses of another class, which is called superclass, by using the extends keyword.

- Hence, we can say that a subclass specializes its superclass.
- Equivalently, one subclass is a special case of the superclass.
- For example, human beings and dogs are two specific types of animals.
- Note that a class can extend only one other class.
- Each superclass has the potential for an unlimited number of subclasses.

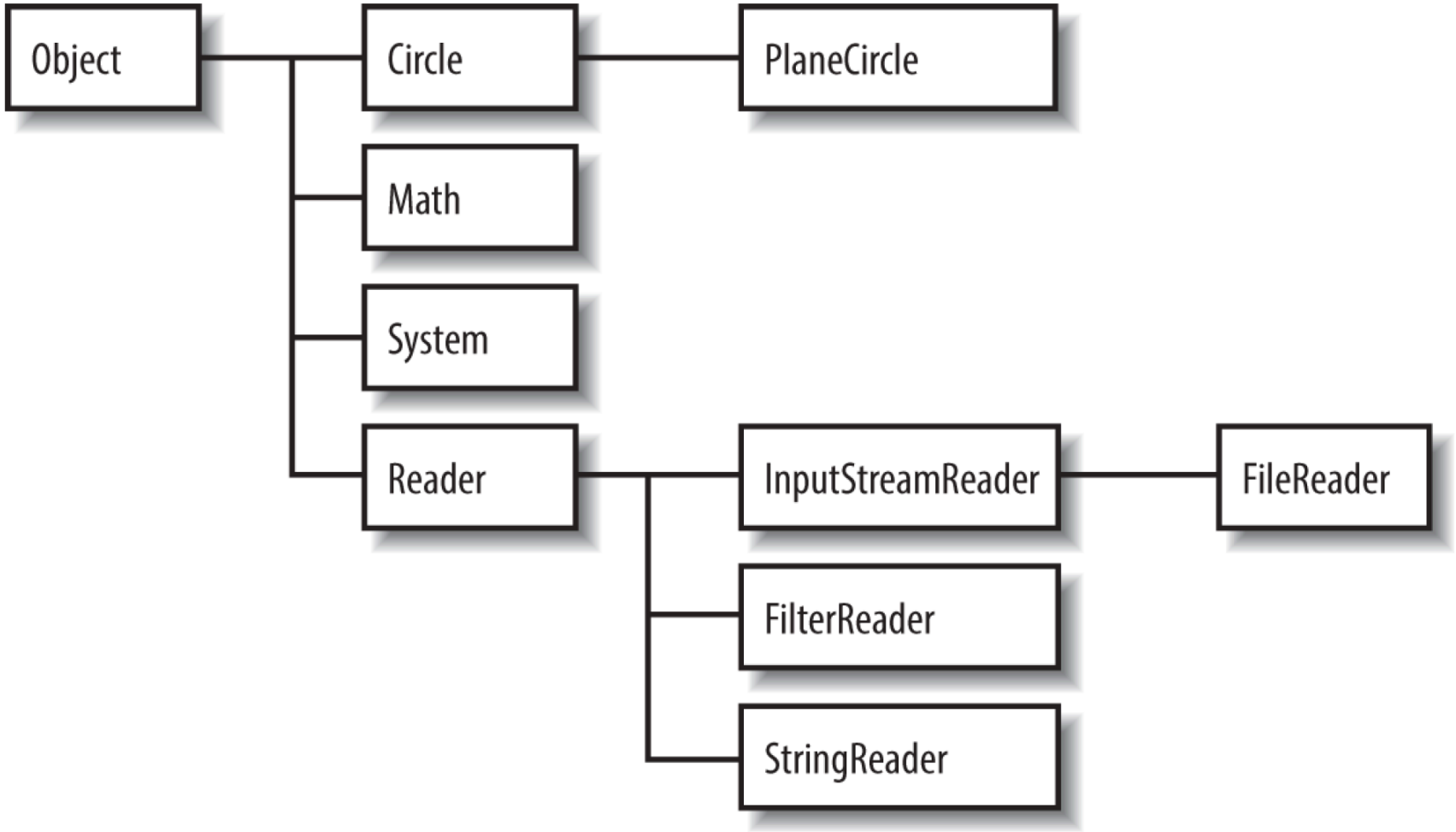
Example

```
class Animal {
    String name;
    int weight;

    Animal(String s, int w) { name = s; weight = w; }

    void eat() { weight += 1; }
    void exercise() { weight -= 1; }
}

class Human extends Animal {
    Human(String s, int w) { super(w, s); }
    void writeCode() { System.out.println("Write codes..."); }
}

class Dog extends Animal {
    Dog(String s, int w) { super(w, s); }
    void watchDoor() { System.out.println("Watch my door..."); }
}
```

The super Keyword

- Recall that the keyword this is used to refer to the object itself.
- You can use the keyword super to refer to members of the superclass.
- Note that this() and super() are used as the constructor of the current class and that of its superclass, respectively.
  - Make sure all the constructor are invoked in the first line in the constructors.

Class Hierarchy[1]



[1] See Fig. 3-1 in p. 113 of Evans and Flanagan.

Constructor Chaining

- If a subclass constructor invokes a constructor of its superclass, you might think that there will be a whole chain of constructors called, all the way back to the constructor of the class **Object**, the topmost class in Java.
- So every class is an immediate or a distant subclass of **Object**.
- Recall that the method **finalize**() and **toString**() are inherited from **Object**.
  - **toString**(): return a string which can be any information stored in the class.

Example

```
class A {
    A() { System.out.println("A is creating..."); }
}

class B extends A {
    B() { System.out.println("B is creating..."); }
    public String toString() {
        return "This is inherited from Object."
    }
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b);
    }
}
```

- The **println**() method can take an object as input, and invoke **toString**() method implicitly.

Field Hiding

- If one field whose name is identical to the field inherited from the superclass, then the newly field hides that of the superclass.
- In other words, the shadowed field of the superclass cannot be referenced by the field name.
- Instead, the field must be accessed through the key word super.

Example

```
class A {
    int x = 1;
}

class B extends A {
    int x = 2;
}

class FieldHidingDemo {
    public static void main(String[] args) {
        B b = new b();
        System.out.println(b.x); // output 2
    }
}
```

Method Overriding

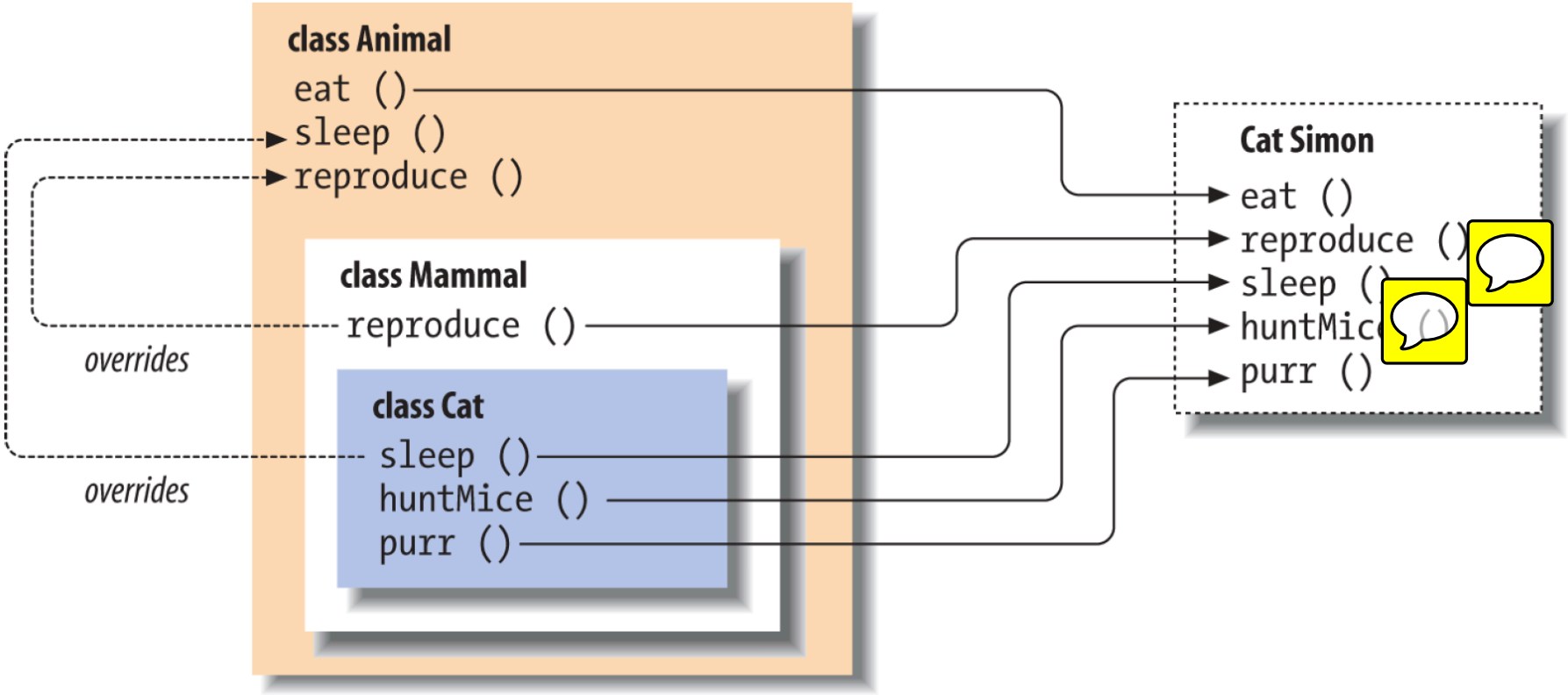
- The subclass is allowed to change the behavior inherited from its superclass if needed.
- As a subclass defines an instance method using the method name, return type, and parameters all identical to the method of its superclass, this method overrides the one of the superclass.[2]
  - Compared to overridden methods, method overloading occurs only in the same class.
- Similarly if your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.

[2] The static methods do not follow this rule.

- When there are multiple implementations of the method in the inheritance hierarchy, the one in the "most derived" class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference variable of the superclass type.[3]

[3] An overridden method in Java acts like a virtual function in C++. See the virtual method table.

Example


class Animal
eat ()
sleep ()
reproduce ()
class Mammal
reproduce ()
class Cat
sleep ()
huntMice ()
purr ()
overrides
overrides
Cat Simon
eat ()
reproduce ()
sleep ()
huntMic
purr ()

Binding

- Association of method definition to the method call is known as binding.
- The binding which can be resolved at compile time by compiler is known as static binding or early binding.
  - They are the static, private and final methods.
  - Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class.
- When compiler is not able to resolve the binding at compile time, such binding is known as dynamic binding or late binding.
  - For example, method overriding.

Polymorphism

- In OO design, the word polymorphism, which literally means "many forms," refers to the ability of reference variables to take different forms.
- The Liskov Substitution Principle states that one variable associated with the declared type can be assigned an instance from any direct or indirect subclass of that type.[4]
- For simplicity, let **U** be a subtype of **T**.

[4] "It is a semantic rather than merely syntactic relation because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular." See `https://en.wikipedia.org/wiki/Liskov_substitution_principle`.

Casting

- Upcasting (widening conversion) is to cast the **U** object to the **T** variable.

```
T t = new U();
```

- Downcasting (narrow conversion) is to cast the **T** variable to a **U** variable.

```
U u = (U) t; // t is T variable reference to a U object.
```

- Upcasting is always allowed, but downcasting is allowed only when the **T** variable refer to a real **U** object.
- This involves type compatibility by JVM during program execution.

instanceof Operator

- The operator instanceof allows us to test whether or not a reference variable is compatible to the object.
- If not compatible, then JVM will throw an exception **ClassCastException**.[5]

[5] We will see the exceptions later.

Example

```
class T {}
class U extends T {}

public class InstanceofDemo {
    public static void main(String[] args) {
        T t1 = new T();

        System.out.println(t1 instanceof U); // output false
        System.out.println(t1 instanceof T); // output true

        T t2 = new U(); // upcasting

        System.out.println(t2 instanceof U); // output true
        System.out.println(t2 instanceof T); // output true

        U u = (U) t2; // downcasting; this is ok.

        u = (U) new T(); // pass the compilation; fail during execution
            !
    }
}
```

Field Hiding with Polymorphism

- You can refer to hidden fields simply by casting an object to a variable of the appropriate superclass.

```
class T {
    int x = 1;
}

class U extends T {
    int x = 2;
}

public class FieldHidingDemo {
    public static void main(String[] args) {
        U u = new U();
        System.out.println(u.x); // output 2
        T t = u;
        System.out.println(t.x); // output 1
    }
}
```

Method Overriding with Polymorphism

- However, you cannot invoke overridden methods by upcasting.
- JVM calls the appropriate method for the object.
  - Method lookup starts from the bottom of the class hierarchy to the top.
  - Always looking for the most specific method body.
- These methods are referred to as virtual methods.
- This mechanism preserves the behaviors of the objects and the superclass type variables play the role of placeholders.

Example

- Imagine that we have a zoo with some animals.

```
class Animal {
    void speak() {}
}
class Dog extends Animal {
    void speak() { System.out.println("woof"); }
}
class Cat extends Animal {
    void speak() { System.out.println("meow"); }
}
class Bird extends Animal {
    void speak() { System.out.println("tweet"); }
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        Animal[] zoo = {new Dog(), new Cat(), new Bird()};
        for (Animal a: zoo) a.speak();
    }
}
```

The final Keyword

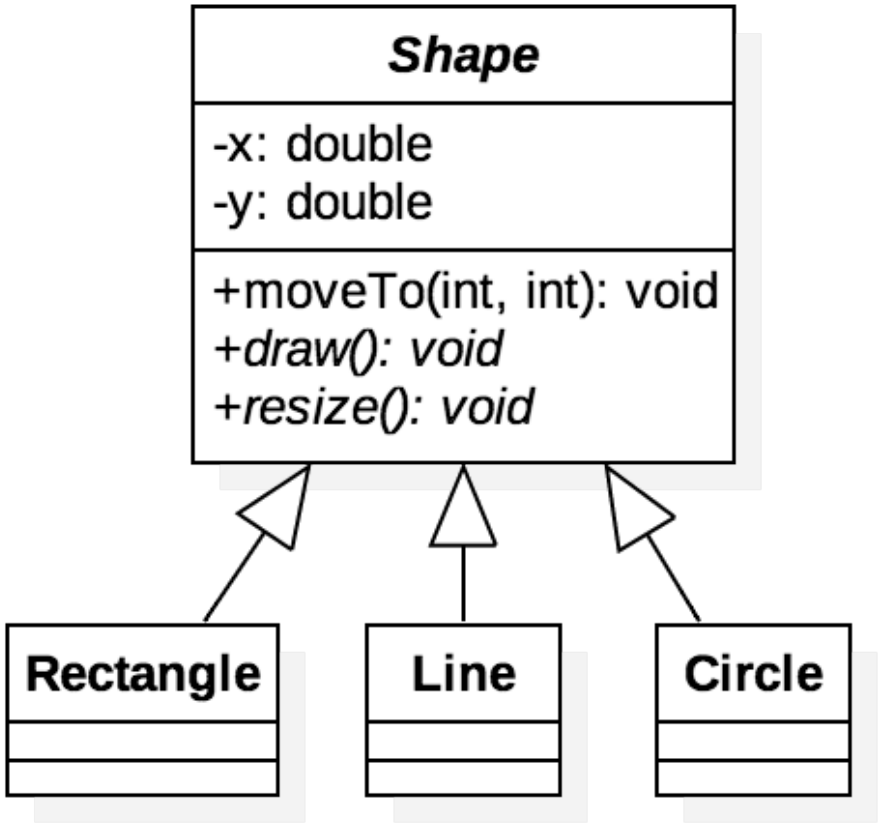
- A final variable is a variable which can be initialized once and cannot be changed later.
  - The compiler makes sure that you can do it only once.
  - A final variable is often declared with static keyword and treated as a constant, for example, **Math.PI**.
- A final method is a method which cannot be overridden by subclasses.
  - You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.
- A class that is declared final cannot be inherited.

Abstract Class

- An abstract class is a class declared abstract.
- The classes that sit at the top of an object hierarchy are typically abstract classes.[6]
- These abstract class may or may not have abstract methods, which are methods declared without implementation.
  - More explicitly, the methods are declared without braces, and followed by a semicolon.
  - If a class has one or more abstract methods, then the class itself must be declared abstract.
- All abstract classes cannot be instantiated.
- Moreover, abstract classes act as placeholders for the subclass objects.

[6] The classes that sit near the bottom of the hierarchy are called concrete classes.

Example



- Abstract methods and classes are in italic.
- In this example, the abstract method *draw*() and *resize*() should be implemented depending on the real shape.

Another IS-A Relationship

- Not all classes share a vertical relationship.
- Instead, some are supposed to perform the specific methods without a vertical relationship.
  - Consider the class **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
  - Both **Bird** and **Airplane** are able to be in the sky.
  - So they should perform the method canFly(), for example.
- By semantics, the method canFly() could not be defined in their superclasses.
- We need a horizontal relationship.

Example

```
interface Flyable {
    void canFly(); // public + abstract
}

abstract class Animal {}

class Bird extends Animal implements Flyable {
    public void canFly() {
        System.out.println("Bird flying...");
    }
}

abstract class Transportation {}

class Airplane extends Transportation implements Flyable  {
    public void canFly() {
        System.out.println("Airplane flying...");
    }
}
```

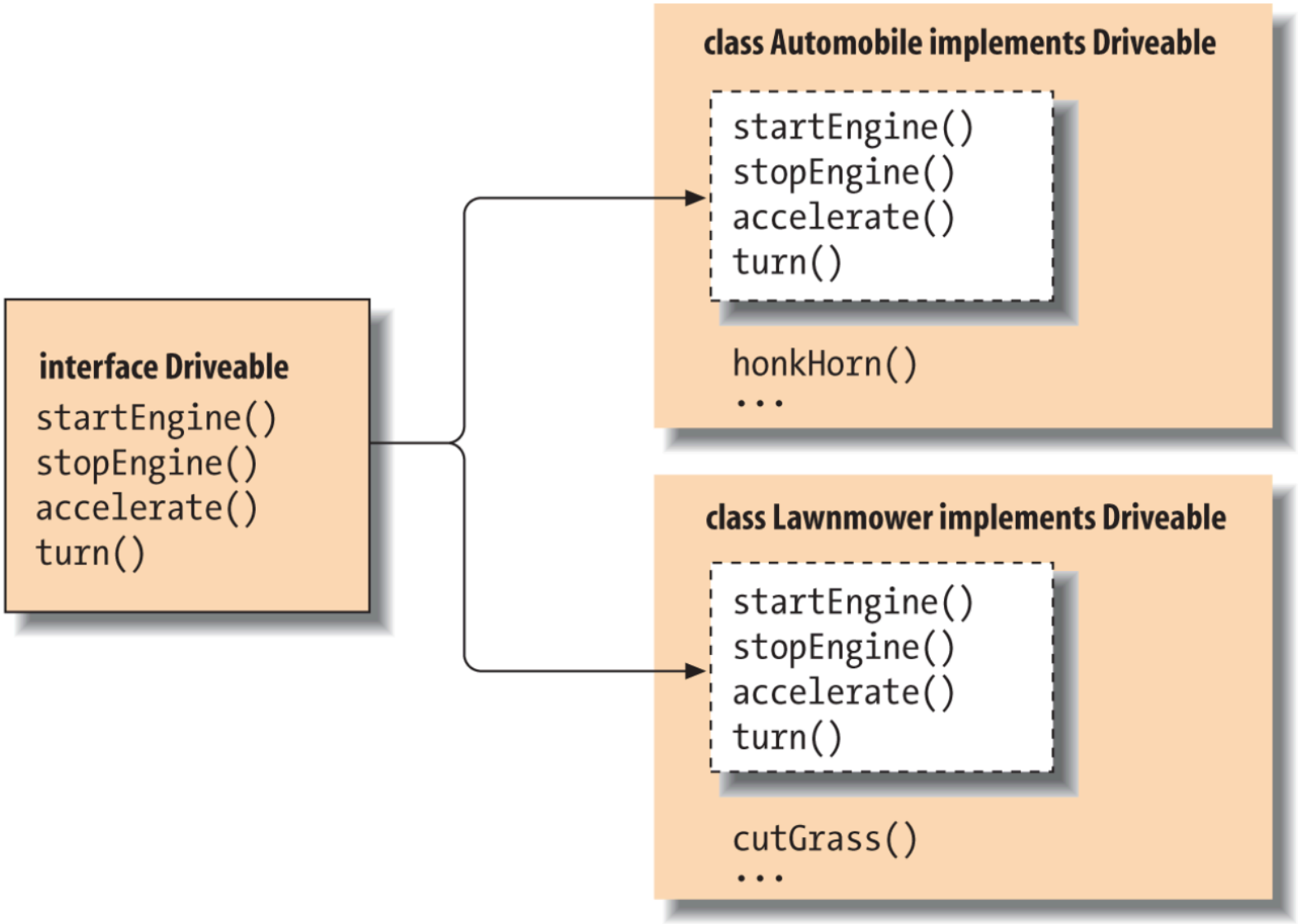
```java
public class interfaceDemo {
    public static void main(String[] args) {
        Airplane a = new Airplane();
        a.canFly();

        Bird b = new Bird();
        b.canFly();

        Flyable f = a;
        f.canFly(); // output ``Airplane flying...''
        f = b;
        f.canFly(); // output ``Bird flying...''
    }
}
```

Interfaces

- An interface forms a contract between the object and the outside world.
  - For example, the buttons on the television set are the interface between you and the electrical wiring on the other side of its plastic casing.
- An interface is also a reference type, just like classes, in which only method signatures are defined.
- So they can be the types of reference variables!

- Note that interfaces cannot be instantiated (directly).
- A class implementing one or multiple interfaces provides method bodies for each defined method signature.
- This allows a class to play different roles, with each role providing a different set of services.
- For example, RPG design.

Example


interface Driveable
startEngine()
stopEngine()
accelerate()
turn()
class Automobile implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
honkHorn()
...
class Lawnmower implements Driveable
startEngine()
stopEngine()
accelerate()
turn()
cutGrass()
...

Properties of Interfaces

- The methods of an interface are implicitly public.
- In most cases, the class which implements the interface should implement all the methods defined in the interface.
  - Otherwise, the class should be abstract.
- An interface can declare only fields which are static and final.
- You can also define static methods in the interface.
- A new feature since Java SE 8 allows to define the methods with implementation in the interface.
  - A method with implementation in the interface is declared default.

- An interface can extend another interface, just like a class which can extend another class.
  - However, an interface can extend many interfaces as you need.
- For example, **Driveable** and **Updateable** are good interface names.
- Common interfaces are **Runnable**[7], **Serializable**[8], and **Collection**[9].

[7] Related to multithreading.

[8] Aka object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

[9] Collections are data structures that are fundamental to all types of programming.

Timing for Interfaces and Abstract Classes

- Consider using abstract classes if any of these statements apply to your situation:
  - share code among several closely related classes
  - declare non-static or non-final fields
- Consider using interfaces if any of these statements apply to your situation:
  - unrelated classes would implement your interface
  - specify the behavior of a particular data type, but not concerned about who implements its behavior
  - take advantage of multiple inheritance