CSCB07 - Software Design

Object Oriented Programming (2)

Summer 2021

Abstract Classes

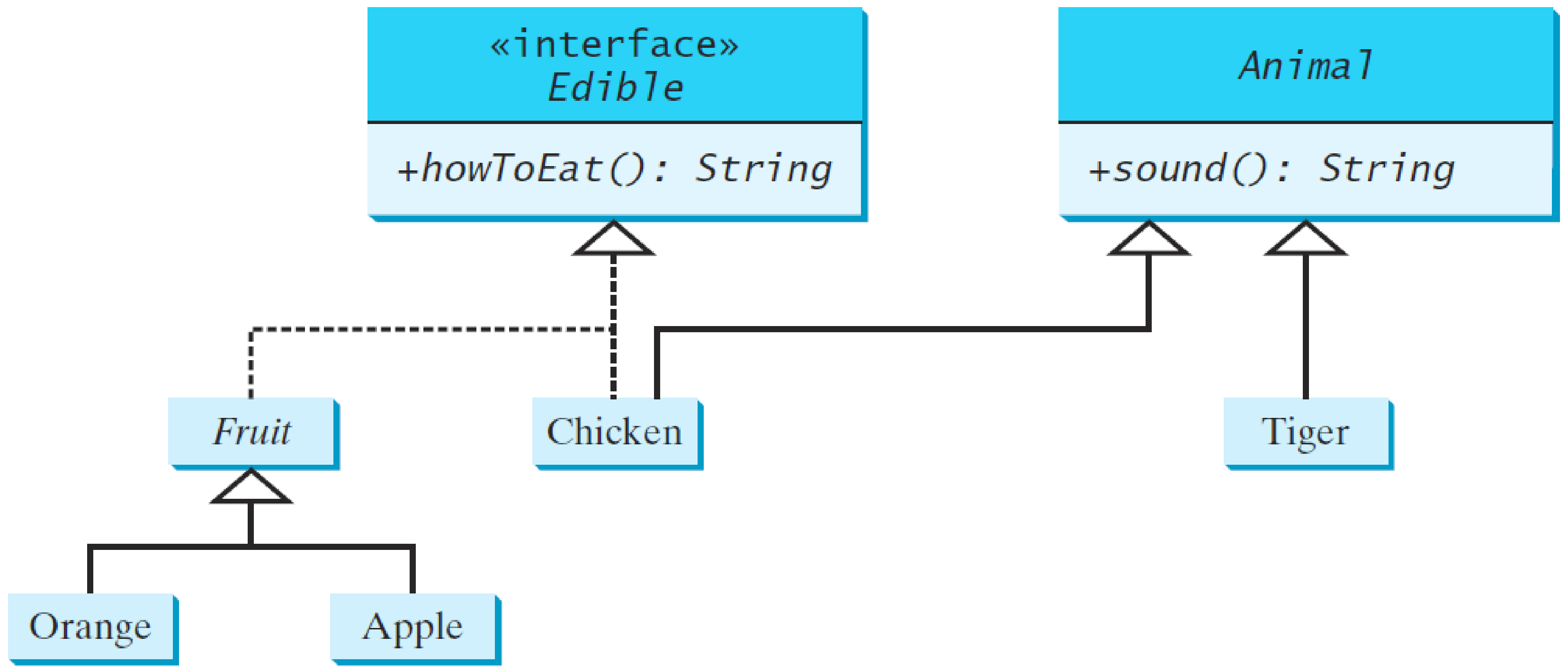
- Cannot be instantiated using the **new** operator
- Usually contain abstract methods that are implemented in concrete subclasses
  - E.g. computeArea() in GeometricObject
- Abstract classes and abstract methods are denoted using the **abstract** modifier in the header
- A class that contains abstract methods must be defined as abstract
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract

Interfaces

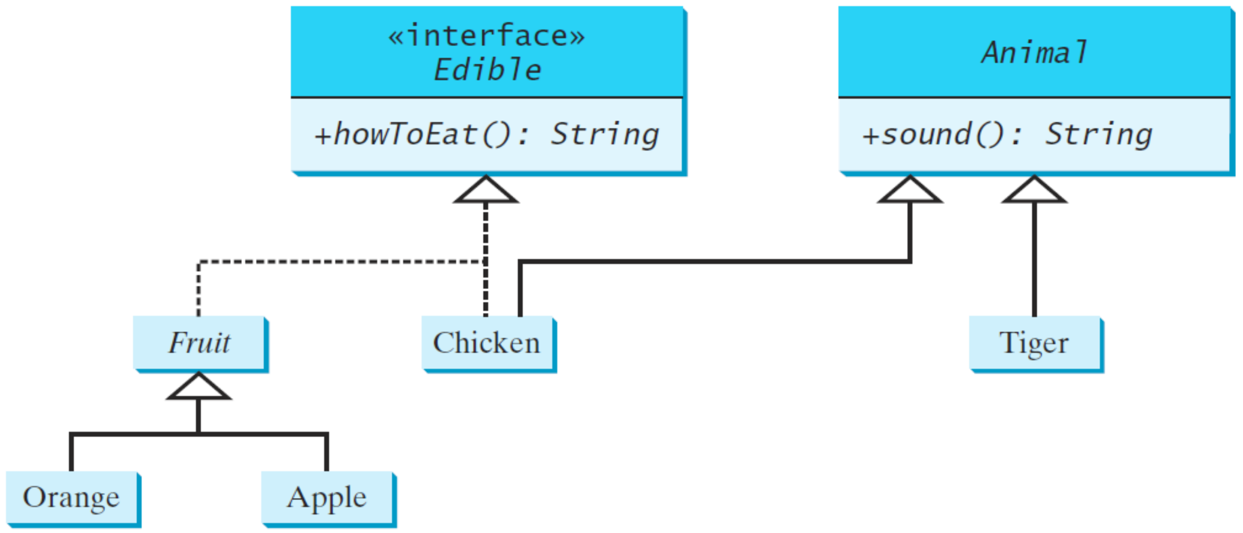
- An interface can be used to define common behaviour for classes (including unrelated classes)
- Contains only constants and abstract methods
- Interfaces are denoted using the **interface** modifier in the header
- Example

```
public interface Edible{
        public abstract String howToEat();
}
```

Interfaces (Example)

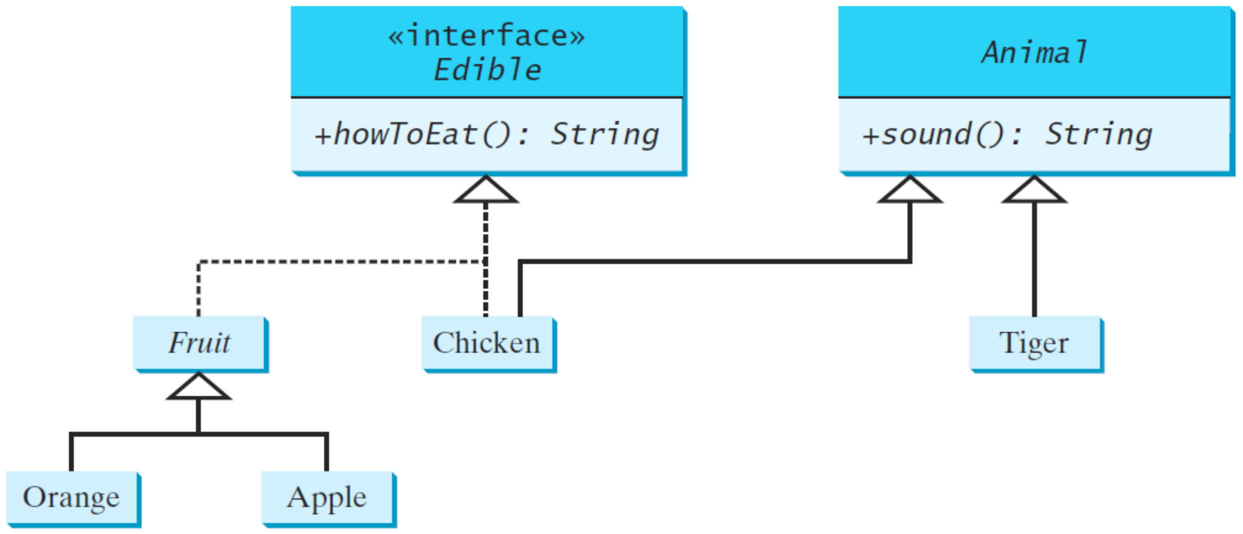


Interfaces (Example)



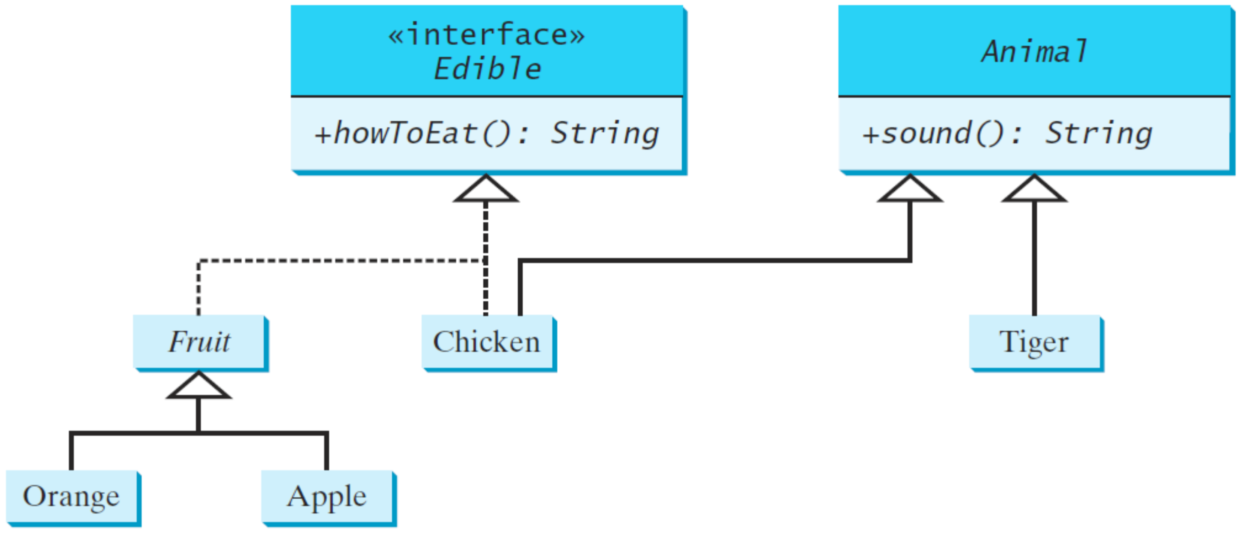
```
abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}
```

Interfaces (Example)



```
class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}
```

Interfaces (Example)



```java
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}

class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

Generics

- Enable type parameterization
  - Generic interfaces
  - Generic classes
  - Generic methods
- Example: **ArrayList** class
  - ArrayList<Integer> A = new ArrayList<Integer>();
  - ArrayList<String> B = new ArrayList<String>();
- Generic types must be reference types
- Enable error detection at compile time

Generics (The **Comparable** interface)

- **Comparable** is a generic interface
  - Defines the **compareTo** method for comparing objects
- Defined as follows:

```
public interface Comparable<T> {
        public int compareTo(T t);
}
```

- The **compareTo** method determines the order of the calling object with **t** and returns a negative integer, zero, or a positive integer if the calling object is less than, equal to, or greater than **t**
- Many classes implement Comparable (e.g. **String, Integer**)

Generics (The **Comparable** interface)

- Implementing Comparable (Example)

```
public class Point implements Comparable<Point> {
        // class body omitted

        @Override
        public int compareTo(Point p) {
                // implementation omitted
        }
}
```

Generics (The ArrayList class)

- Arrays can be used to store lists of objects. However, once an array is created, its size is fixed
- Java provides the generic class **ArrayList** whose size is variable
- Imported using: **import java.util.ArrayList;**
- Commonly used methods (**ArrayList<E>**)
  - **boolean add(E e)**
  - **E get(int index)**
  - **int size()**
  - **boolean contains(Object o)**
  - **int indexOf(Object o)**
- An **ArrayList** could be traversed using a for-each loop

Generics (The HashSet class)

- Generic class that can be used to store elements without duplicates
  - No two elements e1 and e2 can be in the set such that **e1.equals(e2)** is true
- Imported using: **import java.util.HashSet;**
- Objects added to the hash set should override **equals** and **hashCode** properly
- Commonly used methods (**HashSet<E>**)
  - **boolean add(E e)**
  - **int size()**
  - **boolean contains(Object o)**
- A **HashSet** could be traversed using a for-each loop

Generics (The LinkedHashSet class)

- Elements of a **HashSet** are not necessarily stored in the same order they were added
- **LinkedHashSet** is a subclass of **HashSet** with a linked-list implementation that supports an ordering of the elements in the set
- Imported using: **import java.util.LinkedHashSet;**

Exceptions

- Exception handling enables a program to deal with exceptional situations and continue its normal execution
- An exception is an object that represents an error or a condition that prevents execution from proceeding normally
- Exceptions are represented in the **Exception** class, which describes errors caused by the program and by external circumstances
- Developers can create their own exception classes by extending **Exception** or a subclass of **Exception**

Exceptions

- In Java, runtime exceptions are represented in the **RuntimeException** class. Subclasses include:
  - **ArrayIndexOutOfBoundsException**
  - **NullPointerException**
- **RuntimeException** and its subclasses are known as unchecked exceptions
- All other exceptions are known as checked exceptions
  - The compiler forces the programmer to check and deal with them in a try-catch block or declare them in the method header

Exceptions

- Declaring exceptions
  - Every method must state the types of checked exceptions it might throw using the **throws** keyword in the header
  - E.g. **public void myMethod() throws Exception1, Exception2, ..., ExceptionN**
- Throwing exceptions
  - A program that detects an error can create an instance of an appropriate exception type and throw it using the **throw** keyword
  - E.g. **throw new IllegalArgumentException("Wrong Argument");**

Exceptions

- When an exception is thrown, it can be caught and handled in a try-catch block. For example:

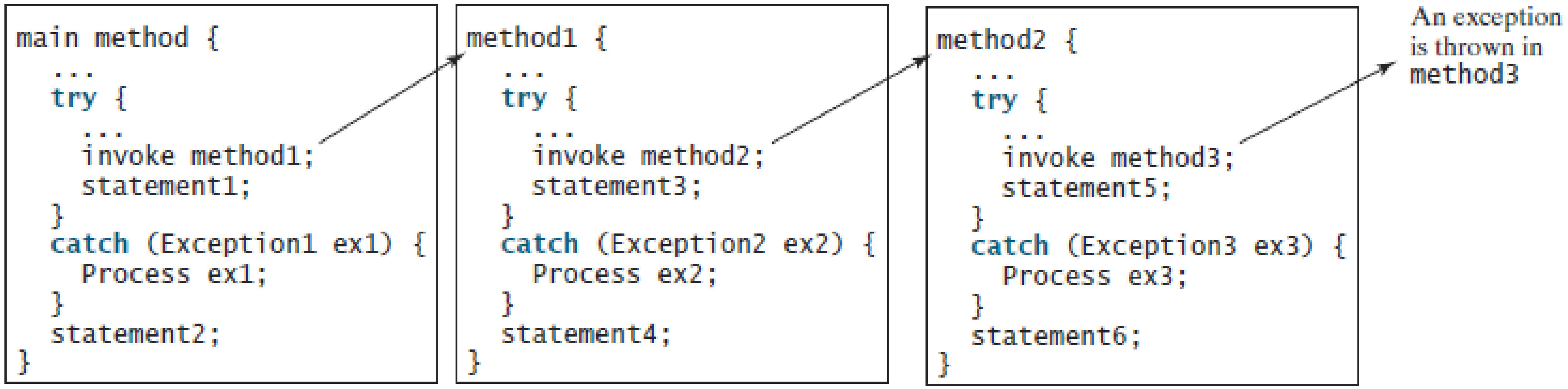
```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

Exceptions

- If no exceptions arise during the execution of the **try** block, the **catch** blocks are skipped
- If one of the statements inside the **try** block throws an exception
  - ➢ The remaining statements in the **try** block are skipped
  - ➢ Each catch block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block
  - ➢ If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler
  - ➢ If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console

Exceptions

- Example



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Exceptions

- Java has a **finally** clause that can be used to execute some code regardless of whether an exception occurs or is caught. For example:

```
try {
        //statements;
}
catch Exception ex) {
        //handling ex;
}
finally {
        //final statements;
}
```

Recommended Reading

- Liang, Introduction to Java Programming, Tenth Edition, © 2015 Pearson Education, Inc.
  - Chapters 12, 13, 19