

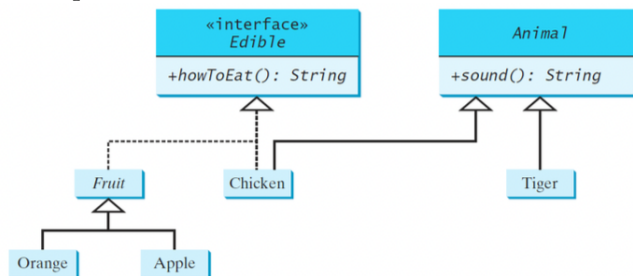
1 Object Oriented Programming (2)

- 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



```
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

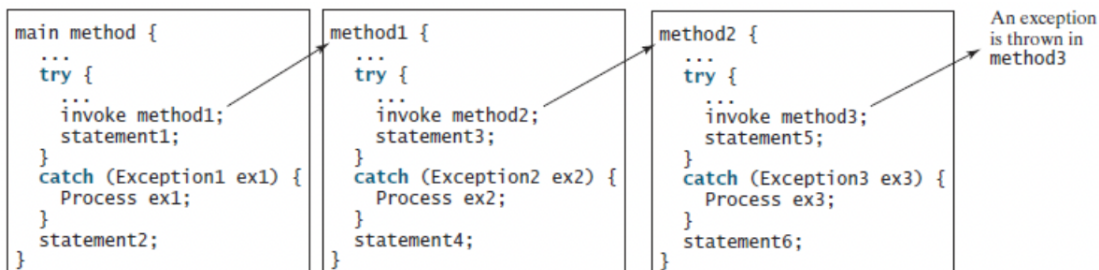
- Enable type parameterization
 - * Generic interfaces
 - * Generic classes
 - * Generic methods
- Example: **ArrayList** class
 - * `ArrayList<Integer> A = new ArrayList<Integer>();`
 - * `ArrayList<String> B = new ArrayList<String>();`
- Generic types must be reference types
- Enable error detection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

- Example



- 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;
}
  
```

Object Oriented Programming - Design Guidelines

- Methods Common to All Objects
 - Always override hashCode when you override equals
 - Always override toString
 - Consider implementing Comparable
- Classes and Interfaces
 - Minimize accessibility
 - Prefer interfaces to abstract classes
 - Favor composition over inheritance
 - Prefer lists to arrays
- Methods
 - Check parameters for validity
 - Return empty arrays or collections, not **null**
 - Document your API properly
- Exceptions
 - Use exceptions only for exceptional conditions
 - Use checked exceptions for recoverable conditions and runtime exceptions for programming errors
 - Do not ignore exceptions
- General Programming (Naming Conventions)
 - Package names should be hierarchical with the components separated by periods. Components should consist of lowercase alphabetic characters and, rarely, digits. e.g. **javax.swing.plaf.metal**
 - Class and interface names should consist of one or more words, with the first letter of each word capitalized
 - Method and field names follow the same typographical conventions as class and interface names, except that the first letter of a method or field name should be lowercase, for example, **ensureCapacity**
 - The names of constant fields should consist of one or more uppercase words separated by the underscore character, for example, **NEGATIVE_INFINITY**
 - Local variable names have similar typographical naming conventions to member names, except that abbreviations are permitted, as are individual characters and short sequences of characters whose meaning depends on the context in which the local variable occurs, for example, **i**, **xref**