CSCB07 - Software Design

Object-Oriented Programming

Fall 2023

Object-Oriented Thinking

- Procedural paradigm
  - ➢ Focuses on designing methods
  - ➢ Data and operations on the data are separate
- Object-oriented paradigm
  - ➢ Couples methods and data together into objects
  - ➢ Organizes programs in a way that mirrors the real world
  - ➢ A program can be viewed as a collection of cooperating objects
  - ➢ Makes programs easier to develop and maintain
  - ➢ Improves software reusability

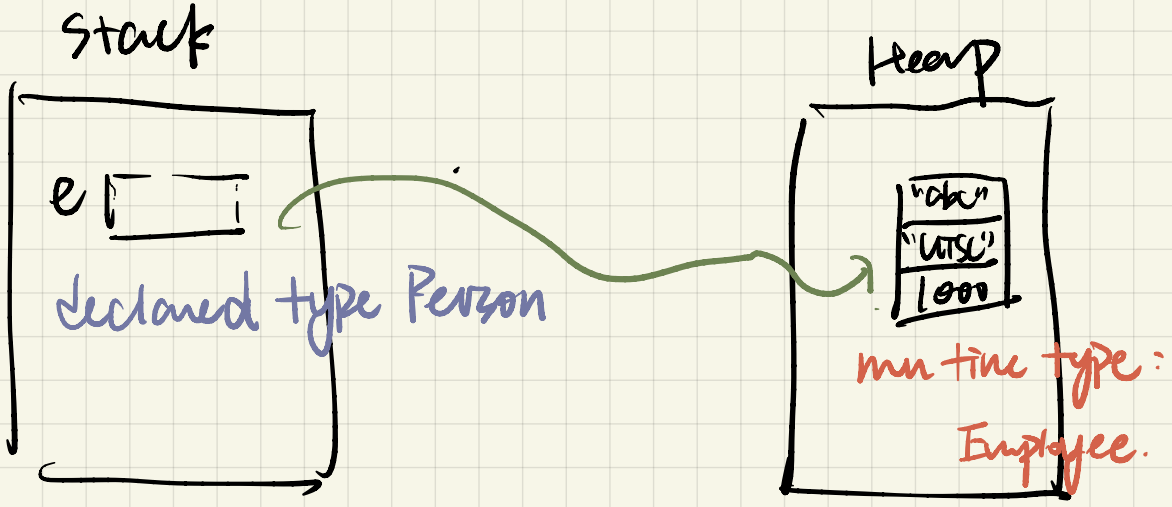
Inheritance

- Powerful feature for reusing software
- Helps avoid redundancy
- Different objects might have common properties and behaviors
  - ➢ E.g. Person, Employee
- Inheritance allows developers to
  - ➢ Define a general class (or superclass). E.g. Person
  - ➢ Extend the general class to a specialized class (or subclass). E.g. Employee
- In Java, the keyword ***extends*** is used to indicate inheritance

Casting objects and the *instanceof* operator

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as upcasting)
  - ➢ E.g. **Person p = new Employee();**
- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used
  - ➢ E.g. **Person p = new Employee(); Employee e = (Employee)p;**
  - ➢ If the superclass object is not an instance of the subclass, a runtime error occurs
  - ➢ It is a good practice to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the ***instanceof*** operator
- Casting an object reference does not create a new object

Person e = new Employee("abc", "UTSC", 1000)



the only thing
that can create
things in the Heap
is using new.

Overloading and Overriding

- Overloading
  - ➢ Defining methods having the same name but different signatures
    - Signature: method name + types of its formal parameters
  - ➢ Overloading methods can make programs clearer and more readable
- Overriding
  - ➢ Defining a method in the subclass using the same signature and the same return type as in its superclass
  - ➢ The ***@Override*** annotation helps avoid mistakes

The *super* keyword

- Refers to the superclass
- Can be used to invoke a superclass constructor
  - ➢ Syntax: ***super***() or ***super***(parameters)
  - ➢ Must be the first statement of the subclass constructor
  - ➢ A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts super() as the first statement in the constructor
  - ➢ If a class is designed to be extended, it is better to provide a no-argument constructor to avoid programming errors
- Can be used to invoke a superclass method
  - ➢ Syntax: ***super*.methodName**(parameters)
  - ➢ Useful in the case of overridden methods

The *Object* class

- Every Java class has ***Object*** as superclass
- It has methods that are usually overridden
  - ***equals*** any object is equal to any other object
  - ***hashCode***
  - ***toString***

The *Object* class: *equals* method

- Header: ***boolean equals(Object obj)***
- The implementation provided by the ***Object*** class checks whether two reference variables point to the same object
  - ➢ Does not check "logical equality"
- When you override the ***equals*** method, you must adhere to its general contract:
  - ➢ *Reflexive*: For any non-null reference value x, x.equals(x) must return true
  - ➢ *Symmetric*: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true
  - ➢ *Transitive*: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true
  - ➢ *Consistent*: For any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false
  - ➢ For any non-null reference value x, x.equals(null) must return false

The *Object* class: *hashCode* method

- Header: ***int hashCode()***
- The implementation provided by the ***Object*** class returns the memory address of the object
- The ***hashCode*** method should be overridden in every class that overrides ***equals***
  - ➢ Equal objects must have equal hash codes
- A good ***hashCode*** method tends to produce unequal hash codes for unequal objects

The *Object* class: *toString* method

- Header: ***String toString()***
- The ***toString*** method is automatically invoked when an object is passed to ***println*** and the string concatenation operator
- Class ***Object*** provides an implementation of the ***toString*** method that returns a string consisting of the class name followed by an "at" sign (@) and the unsigned hexadecimal representation of the hash code
- ***toString*** is usually overridden so that it returns a descriptive string representation of the object

Polymorphism

- Every instance of a subclass is also an instance of its superclass, but not vice versa
- Polymorphism: An object of a subclass can be used wherever its superclass object is used
- Example

```
public class Demo {
    public static void main(String [] args) {
        m(new Point(1,2));
    }

    public static void m(Object x) {
        System.out.println(x);
    }
}
```

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain
- The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable
  - ➢ E.g. **Object x = new Point(1,2);** //declared type: Object, actual type: Point
    System.out.println(x); //which toString is invoked?
- Dynamic binding works as follows:
  - ➢ Suppose an object x is an instance of classes C1, C2, . . . , Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, . . . , and Cn-1 is a subclass of Cn,
  - ➢If x invokes a method p, the JVM searches for the implementation of the method p in C1, C2, . . . , Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked

Encapsulation

- The details of implementation are encapsulated and hidden from the user
- Modules communicate only through their APIs and are oblivious to each others' inner workings
  - ➢ E.g. using ***System.out.println*** without knowing how it is implemented
- Advantages
  - ➢ Decoupling the modules that comprise a system allows them to be developed, tested, optimized, used, understood, and modified in isolation
  - ➢ Information hiding increases software reuse because modules that aren't tightly coupled often prove useful in other contexts

Encapsulation

- The access control mechanism in Java facilitates encapsulation
- There are four possible access levels for members, listed in order of increasing accessibility:
  1) **private**—The member is accessible only from the top-level class where it is declared
  2) **package-private**—The member is accessible from any class in the package where it is declared (default access)
  3) **protected**—The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared
  4) **public**—The member is accessible from anywhere
- Rule of thumb: *make each member as inaccessible as possible*

Syntax Error

Abstract Classes

use "abstract class" key word.
avoid redundant code.

- 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

for (Object obj : stuff) { ... }
instead of for (int i=0; i< stuff.length; i++) {
Object obj = stuff[i]; }

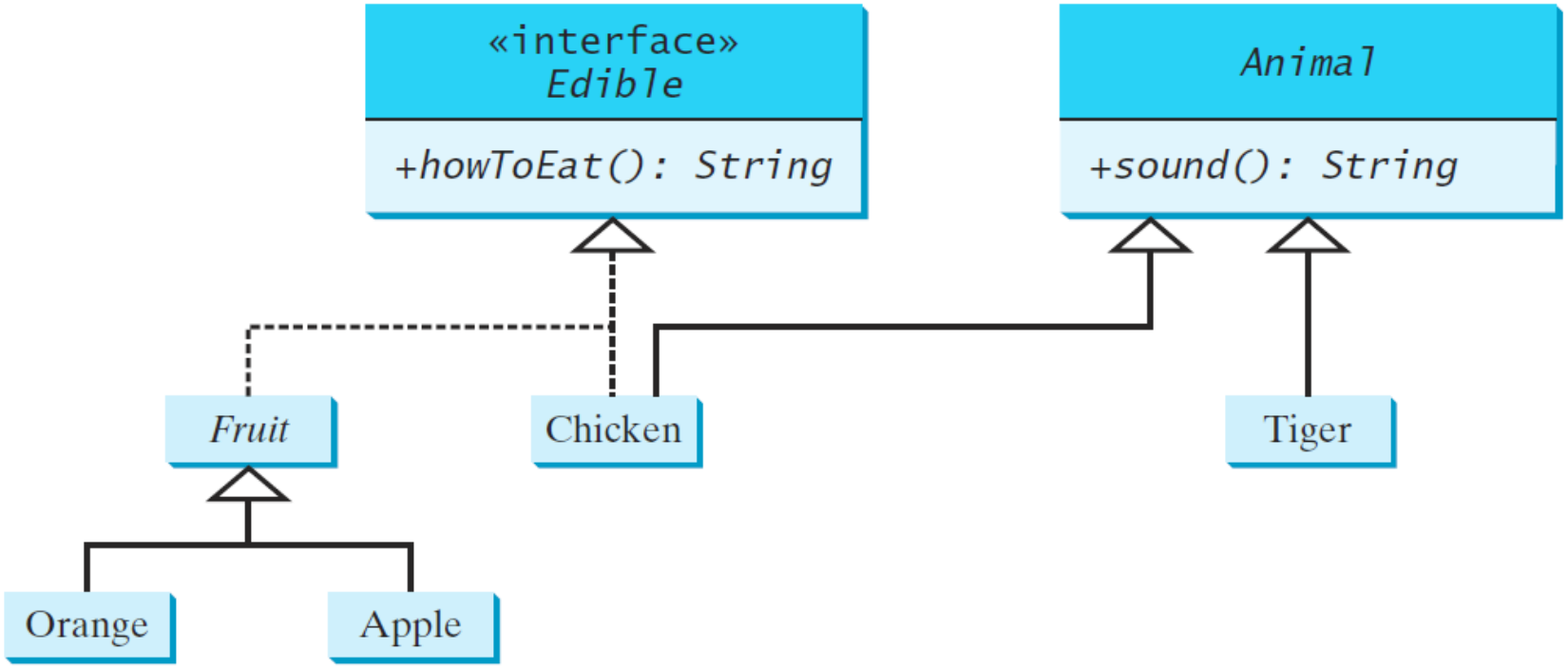
- An interface can be used to define common behaviour for classes (including unrelated classes)
- Contains only constants and abstract methods
- Interfaces are denoted using the **interface** modifier in the header
- Example

not a class ⇒ no need for difference
abstract keyword.

interface
abstract class

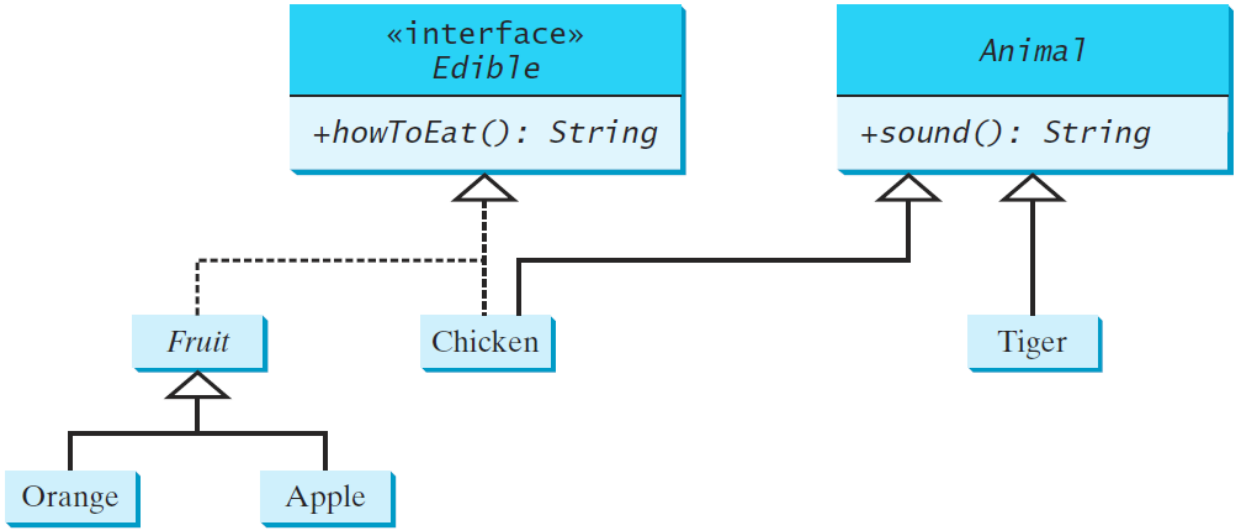
```
public interface Edible{
        public abstract String howToEat();
}
```

Interfaces (Example)



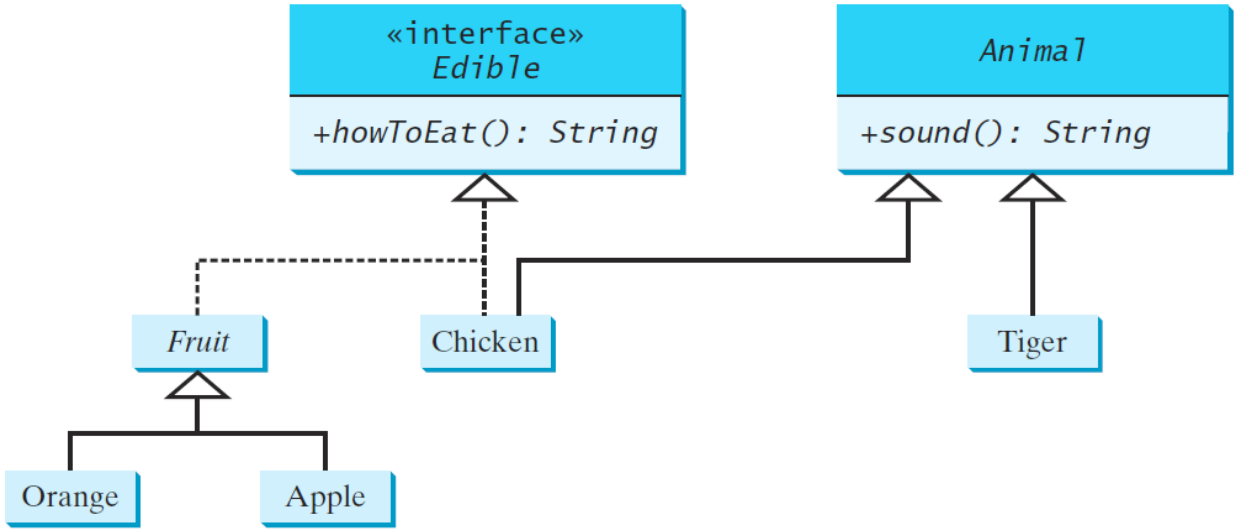
Liang, Introduction to Java Programming, Tenth Edition, © 2015 Pearson Education, Inc.
Bloch, Effective Java, Second Edition, © 2008 Sun Microsystems, Inc.

Interfaces (Example)



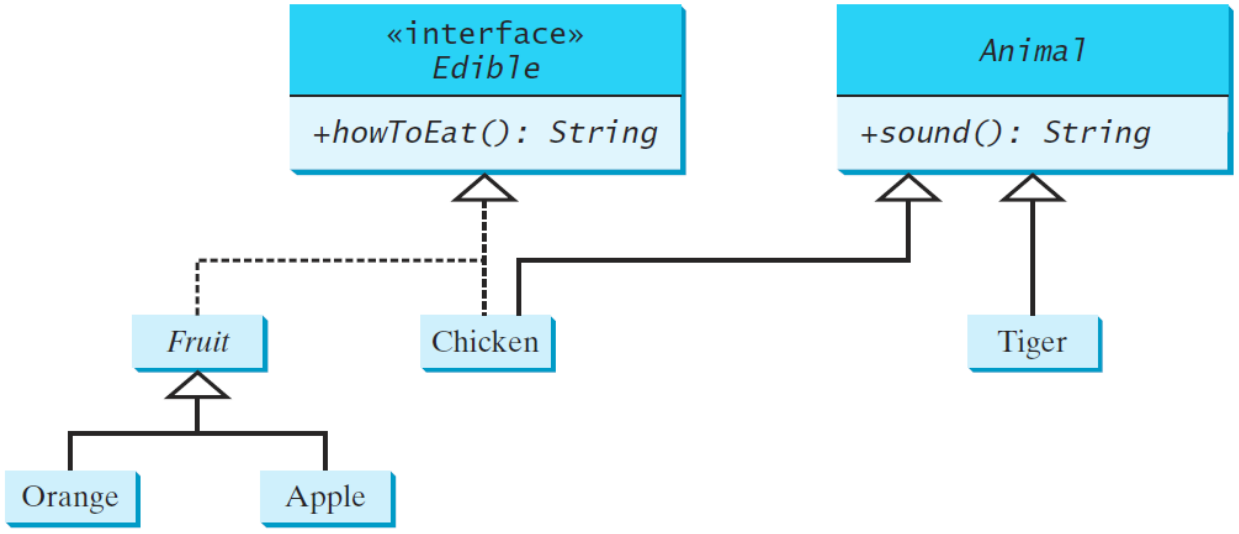

```
abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

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)




```
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
- Generic types could be bounded using the **extends** keyword

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

runtime exception
↳ do not need to handle it
check exception
↳ need to handle it.
try catch.

- 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:

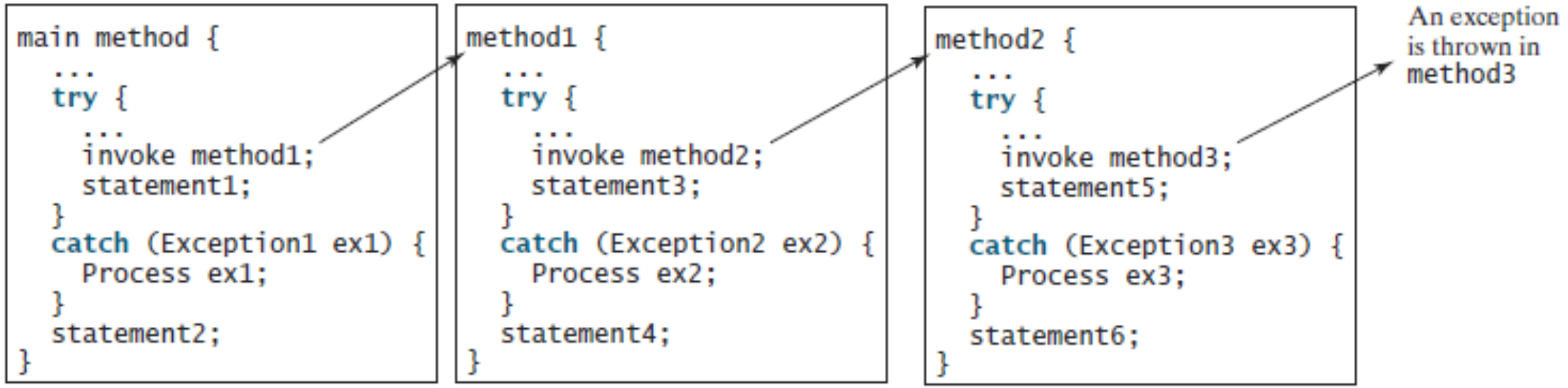
```
try {
  statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

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



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;
}
```