Chapter 5

Object-Oriented Programming (2)

References :

[1] รังสิพรรณ มฤคทัต, กระบวนทัศน์ในการเขียนโปรแกรม (บทที่ 4-5)

[2] Tucker & Noonan, Programming Languages: Principles and Paradigms (Chapters 7, 13)

[3] Dietel & Dietel, Java: How to program (Chapters 9, 14)

[4] Oracle, Java Documentation

Chapter Objectives

At the end of this chapter, you should be able to:

- Compare abstract classes and interfaces
- Explain generic classes and type parameters
- Explain exception propagation and exception handling of checked and unchecked exceptions
- Hand trace Java programs with abstraction & exceptions
- Write Java programs with interfaces (e.g. Comparable) and generic classes (e.g. ArrayList)
- Write Java programs that are robust to exceptions

Effective OOP

- High-level abstraction design : large system is specified at a very general level, long before the detailed implementation takes place
  - Abstract Class
  - Interface
- Minimal programming : eliminate redundancy
  - Template / Generic
- Exception handling : ensure program is robust

Abstract Class

```
abstract class A {
   abstract public void f1 (..);
   public void f2 (..) { .. }
}
```

A pa; ✓

new A (); ✗

- Abstract method – no method body, just header
- Abstract class
  - If a class contains any abstract method, it must be declared an abstract class
  - An abstract class may or may not have abstract methods
  - Subclass that has not implemented all inherited abstract methods is itself an abstract class

Abstract Class and Polymorphism

```
abstract class Anyshape                    { abstract public void draw(); }
class MyCircle       extends Anyshape      { public void draw() { … } }
class MyRectangle    extends Anyshape      { public void draw() { … } }
class MyTriangle     extends Anyshape      { public void draw() { … } }
```

```
Anyshape parent[ ] = new Anyshape[3];
parent[0] = new MyCircle();
parent[1] = new MyRectangle();
parent[2] = new MyTriangle();

for (int i = 0; i < 3; i++)  parent[ i ].draw()
```

Force the implementation of draw() in subclasses

Anyshape cannot be used to create objects
But it can be used to create an array (of references to objects)

Limitting Class Usage (Revisit)

- Class with only private constructors
  - Cannot create object (outside class) because client code cannot access private methods
  - Cannot pass its properties to subclasses (inheritance) because, along the constructor chain, the subclass cannot access private constructors
- Abstract class
  - Can declare reference variable, but cannot create object (there are abstract methods that cannot be executed)
  - Can pass its properties to subclasses
  - Typically used as a base class when we need to force subclasses to implement certain concepts

Interface

```
interface InA {
    static final int x      =  1;
    static final String s   =  “Hello”;
    public void f ( );
}
class A implements InA { … }
```

- Separate logical design from implementation detail
  - Member methods and vars are implicitly public
  - Methods are implicitly abstract & non-static
  - Vars are implicitly static & final (must be initialized)
  - No constructor in the interface

Interface *vs.* Abstract Class

- Interface
  - Purely describe key concepts, i.e. methods & constants, without implementation details
  - Object creation : have a class implementing this interface and create objects from this class
- Abstract class
  - Base class with key variables and methods
  - Some methods may be fully implemented
  - Object creation : have a class extending this abstract class and create objects from subclass

Implementation of Interface

- A class is concrete, i.e. allows objects to be created, only if it implements all methods in the interface
  - Otherwise, it is only an abstract class
  - Vars or methods other than those declared in the interface may be added
  - A class may implement multiple interfaces
    class A implements In1, In2 { … }
  - Multiple concrete classes may implement the same interface differently ➔ facilitate polymorphism

Implement abstract method ➔ add method body and remove prefix abstract

Interface and Polymorphism

```
interface Anyshape                                  { public void draw();  }
class MyCircle      implements  Anyshape    { public void draw()  { … } }
class MyRectangle   implements  Anyshape    { public void draw()  { … } }
class MyTriangle    implements  Anyshape    { public void draw()  { … } }
```

```
Anyshape  parent[ ]  =  new Anyshape[3];
parent[0]  =  new MyCircle();
parent[1]  =  new MyRectangle();
parent[2]  =  new MyTriangle();

for (int i = 0; i < 3; i++)  parent[ i ].draw()
```

Anyshape cannot be used to create objects
But it can be used to create an array (of references to objects)

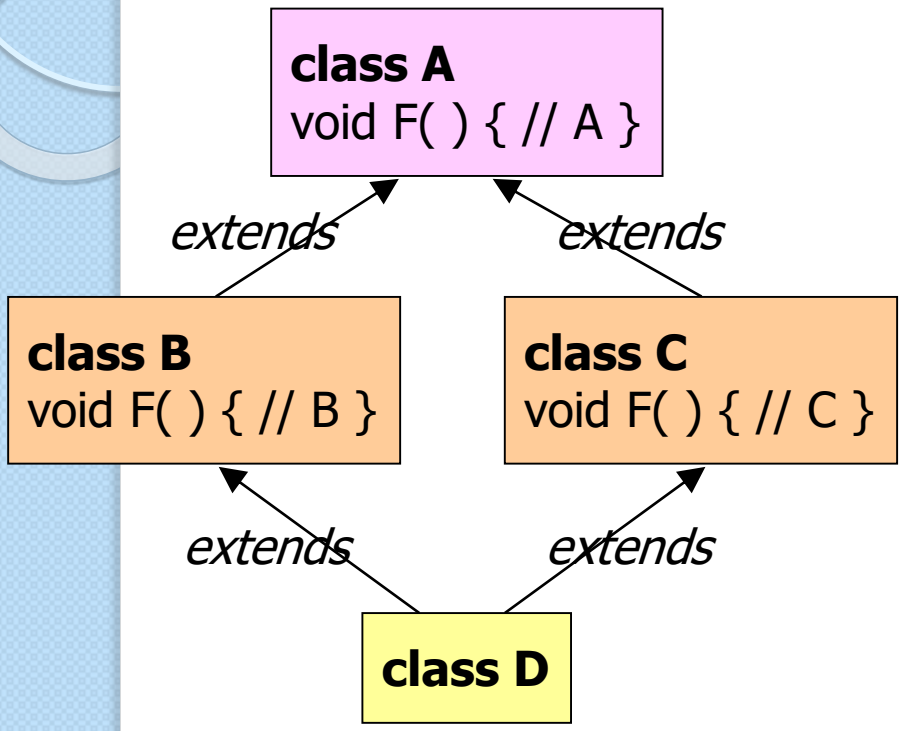
Interface and Inheritance

```
interface InA
{
    public static final int x = 1;
    public void f( );
    public void g( );
}
```

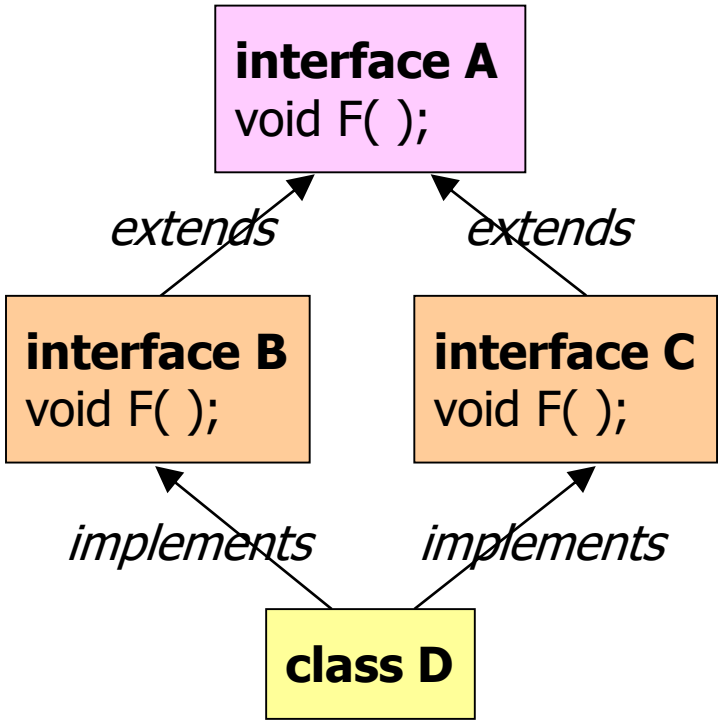
```
interface InB extends InA
{
    public static final int x = 2;
    public void f( );
    public void h( );
}
```

- class MyClass implements InB
  - MyClass gets 2 copies of **x** : InA.x = 1, InB.x = 2
  - But it can have only 1 implementation of **f( )**
    - InA.f( ) and InB.f( ) are considered the same method (they are only method headers without any details)

Diamond Inheritance



Not allowed in Java
(avoid collision between diff versions of concrete methods)



Allowed in Java
(abstract method is implemented just once)

Generic Method (Method Template)

- Allow one method to accept any type of parameters

```
class A {
    public <T, V> void test (T  t, V  v)  { … }
}
A pa = new A( );
pa.test("Hello", 10);        // T = String, V = Integer
pa.test(10, "Hello");        // T = Integer, V = String
```

- Generic declaration <T, V> is added to method header (before return type), telling compiler that they are type parameters, not type name
- Compiler converts T and V to java.lang.Object, so they can support any type

Type Name *vs.* Type Parameter

- Specific type name

```
class S { … }
class B {
  public void test(S s) {
     S var;                ✓
     var = new S( );       ✓
     …
  }
}
```

- **S** is a user-define type

- Type parameter

```
class A {
   public <T> void test (T  t) {
      T var;                ✓
      var = new T( );       ✗
      …
   }
}
```

- **T** means any type (must be class, not primitive type)

Generic Class (Class Template)

- A class containing non-static variables whose types can be any type must be declared generic class

```
class A <T, V>  {
    private  T  t;
    private  V  v;
    A(T  ti, V  vi)                    { … }
    public void test (T  t, V  v)      { … }
}
```

  - Generic declaration <T, V> is added to class header. So it is not needed in method header
  - Compiler converts T and V to java.lang.Object

```
A <Integer, String> p1   =  new A <Integer, String> (1, "A");
A <String, Integer> p2   =  new A <String, Integer> ("A", 1);

p1.test(10, "Hello");      // <T, V> is specifically <Integer, String>
p2.test("Hello", 10);      // <T, V> is specifically <String, Integer>
```

- When creating object, we must specify actual types or type arguments in < >
  - T and V are java.lang.Object, so they support any type
- Once actual types are specified, correct type of data must be sent to method in subsequent method calls

Bounded Type Parameter

- class A **<T>** { … }
  - **T** supports any type that inherits from Object
  - A <Number> p1 = new A <Number> (…); ✓
  - A <Integer> p2 = new A <Integer> (…); ✓
  - A <String> p3 = new A <String> (…); ✓
- class A **<T extends Number>** { … }
  - **T** supports any type that inherits from Number
  - A <Number> p1 = new A <Number> (…); ✓
  - A <Integer> p2 = new A <Integer> (…); ✓
  - A <String> p3 = new A <String> (…); ✗

Inherit from Generic Class

- To extend class A <T, V>, subclass may
  - Keep type parameters from parent
  - Add more type parameters
  - Give specific types to type parameters from parent

class B <T, V, W> extends A <T, V>	new B <String, Float, Integer> ( );
class B <T> extends A <T, Float>	new B <String> ( );
class B extends A <String, Float>	new B ( );
class B <W> extends A <String, Float>	new B <Integer> ( )
class B extends A same as extends A <Object, Object>	new B ( );

- To create object from B ➔ count #type parameters in < > and specify actual type to each type parameter

Generic Class with Different Type Arguments

- class A <T> { … }
  - A <Number> p1 = new A <Number> (…);
  - A <Integer> p2 = new A <Integer> (…);
  - A <String> p3 = new A <String> (…);
- p1, p2, p3 are from the same generic class but with different type arguments ➔ different types
  - p1 = p2; ✘
  - p1 = new A <Integer> (…); ✘

Wildcard (?)

- Mean any type argument when declaring reference var
- But cannot be used with object creation
- class A **<T>** { … }
  - A <Number> p1 = new A <Number> (…);
  - A <Integer> p2 = new A <Integer> (…);
  - A <String> p3 = new A <String> (…);

Any type argument that inherits from	
Object	Number
A<?> po; po = p1; ✓ po = p2; ✓ po = p3 ✓	**A<? extends Number>** pn; pn = p1; ✓ pn = p2; ✓ pn = p3 ✗

Java Collections Framework

- import java.util.*;
- Ready-made classes that represent data structures and basic algorithms (search, sort, min, max, etc.)
- Most of them are in generic form

```
// keep objects from class String
ArrayList<String> AL =  new ArrayList<String>();
AL.add("Anna");  AL.add("Betty");
```

Class java.util.ArrayList

- Extendable array
- E = type of data to be kept, must be class name
  - public ArrayList<E> ( )
  - public ArrayList<E> (ArrayList<E> other)
  - public void add (E data) // append
  - public void add (int index, E data) // insert at index
  - public void clear ( )
  - public E remove (int index) // return removed data
  - public int size ( )
  - public boolean isEmpty ( )
  - public E get (int index) // return data at index

- java.util.Arrays : handle data in array
- java.util.Collections : handle data in other types of containers such as Vector, LinkedList, etc.

```
int [] A = {30, 10, 20};
Arrays.sort(A);              // sore array
Collections.sort(AL);        // sort ArrayList
```

- Search, sort, min, max with user-defined class
  - The class must implement interface Comparable with method compareTo(Object param) that returns

-1	if this object < param object
0	if this object = param object
1	if this object > param object

```
class MyPoint implements Comparable {
    private int x, y;
    public MyPoint(int a, int b)   { x = a; y = b; }

    // rules for comparing this and another MyPoint object
    public int compareTo(Object param) {
        MyPoint other = (MyPoint) param;
        if (this.x < other.x)          return -1;
        else if (this.x > other.x)     return 1;
        else                           return 0;
    }

    public void print()    { … }
}
```

sorting depends on value of x only

```
MyPoint [] P = new MyPoint[4];
P[0] = new MyPoint(40, 100);
P[1] = new MyPoint(10, 400);
P[2] = new MyPoint(30, 200);
P[3] = new MyPoint(20, 300);
```

```
Arrays.sort( P );
for (int i = 0; i < P.length; i++)
    P[i].print( );
```

```
ArrayList <MyPoint> AL = new ArrayList <MyPoint> ( );
for (int i = 0; i < P.length; i++)  AL.add( P[i] );
Collections.sort( AL );
for (int i = 0; i < AL.size(); i++)  AL.get(i).print( );
```

```
(10, 400)
(20, 300)
(30, 200)
(40, 400)
```

objects are sorted by x-value as specified by compareTo

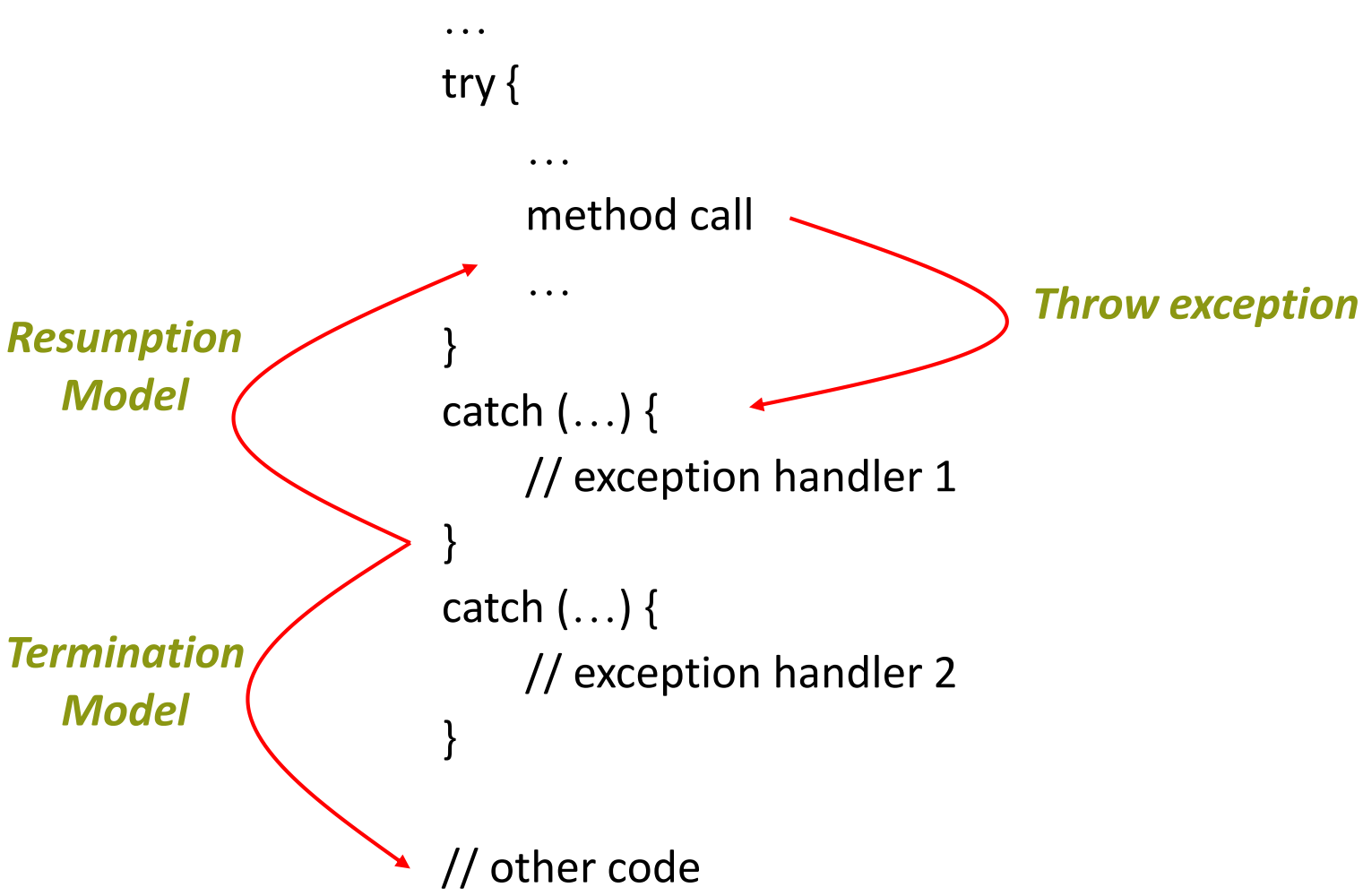
Exception

- Error condition occurring from an operation that cannot be resolved by the operation itself
- Exception mechanism
  - Exceptional condition
  - Offending command which causes exception (raise or throw exception)
  - Exception handler which handles exception (catch exception)
- Robust program should be able to continue even when exceptions occurs
  - Fortran, Pascal, C : programmer's own responsibility
  - Ada, C++, Java : exception mechanisms are provided

Exception Model

- Issue 1 → how are handlers associated with exceptions
  - Hardware-level : usually 1 handler for each specific exception
  - Application-level : different portions of program may respond differently to a specific exception
- Issue 2 → how an exception handler is located
  - Exception usually propagates outwards from the block where the offending command resides
- Issue 3 → how to continue after the exception is handled
  - Termination model : offending code cannot be resumed (Ada, C++, Java)
  - Resumption model : flow of control resumes at the code immediately after offending code (PL/I)

Exception Handling

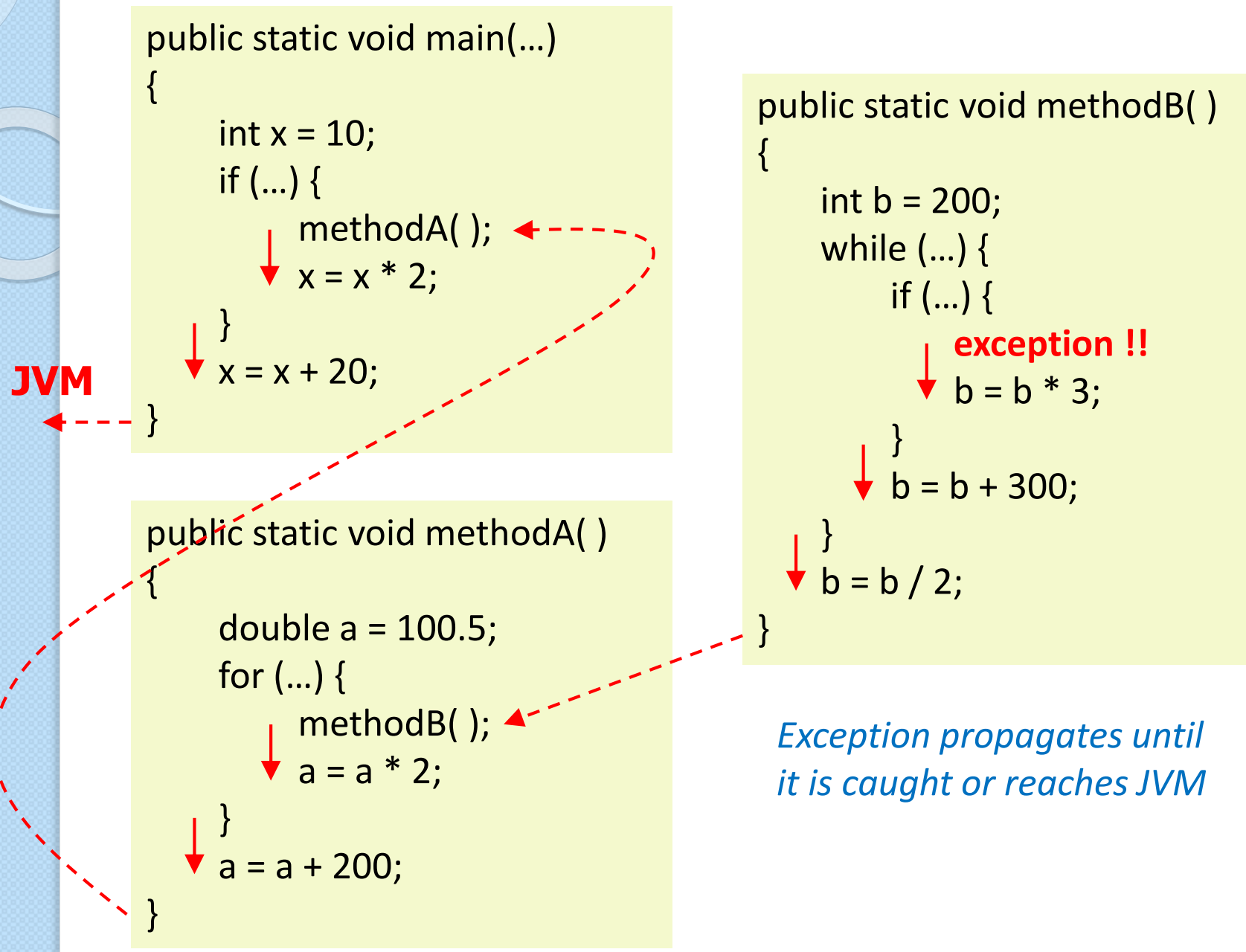


Java Exception Mechanism

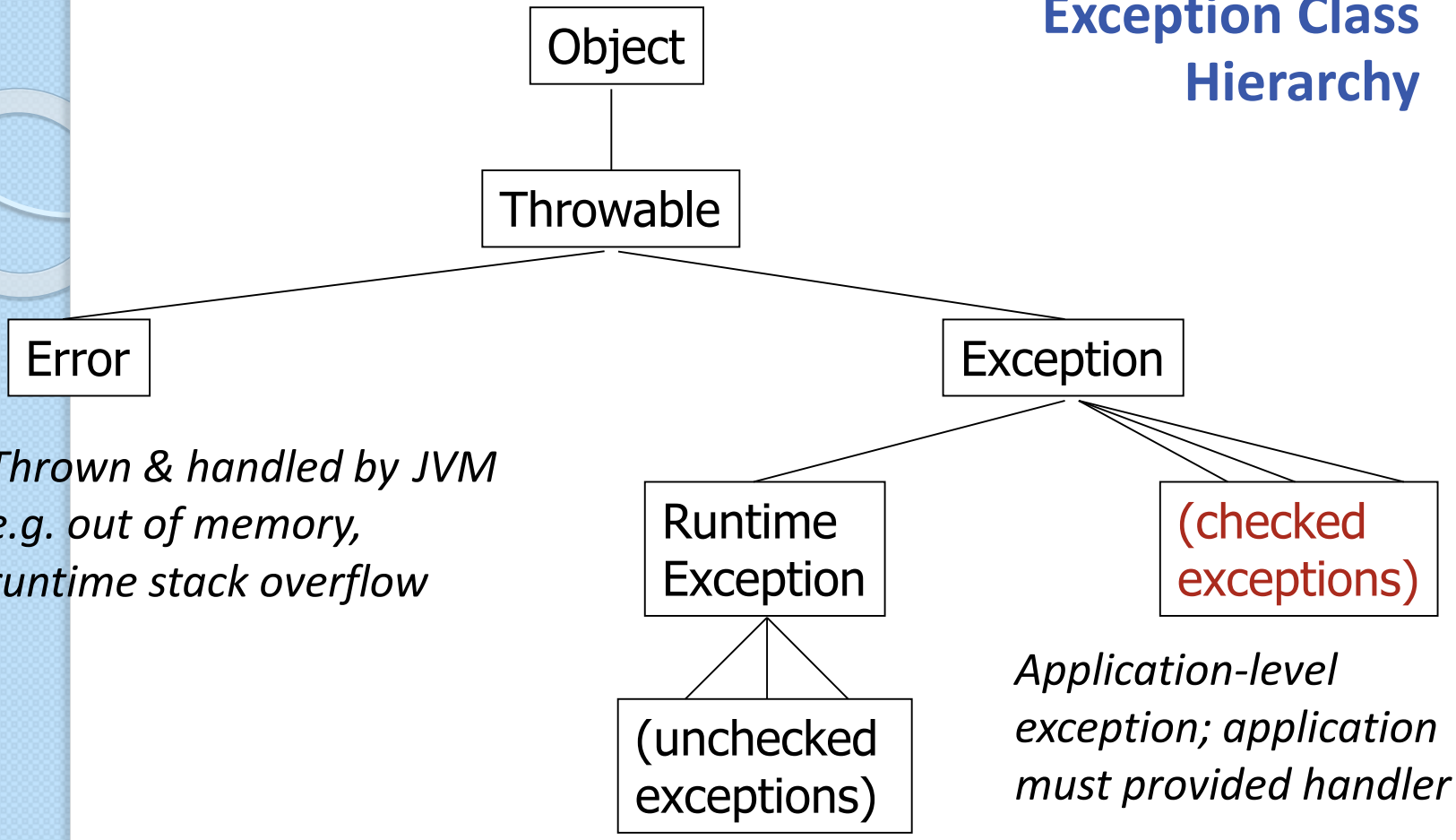
```
try {
      // do something that might throw exception
}
catch ( exception object 1 )    {  // exception handler 1  }
catch ( exception object 2 )    {  // exception handler 2  }
finally  {  // do something before leaving  }

// other code
```

- Exception propagates downwards & outwards until it is caught or out of method main()
  - Once it is out of main(), it will be handled by JVM


public static void main(…)
{
int x = 10;
if (…) {
methodA( );
x = x * 2;
}
x = x + 20;
}
JVM
public static void methodA( )
{
double a = 100.5;
for (…) {
methodB( );
a = a * 2;
}
a = a + 200;
}
public static void methodB( )
{
int b = 200;
while (…) {
if (…) {
exception !!
b = b * 3;
}
b = b + 300;
}
b = b / 2;
}
Exception propagates until
it is caught or reaches JVM

Exception Class Hierarchy



Thrown & handled by JVM
e.g. out of memory,
runtime stack overflow

Application-level exception; application must provided handler

Application is not obliged to provide handler; but it will crash if exception occurs & is not handled

Checked Exception

- Class **Exception**
  - public Exception ( )
  - public Exception (String message)
  - public void printStackTrace ( )

Cannot propagate out of method by itself (must be thrown in method header)

```
// user-defined exception
class MyException extends Exception {
    public MyException ( );
    public MyException (String message)        {  super(message);  }
}
// throwing exception
throw new MyException("My message");                              // (1)

MyException myExceptionObject = new MyException();        // (2)
throw myExceptionObject;
```

Unchecked Exception

- Class **RuntimeException** extends Exception

```
// user-defined exception
class MyRuntimeException extends RuntimeException {
    public MyRuntimeException ( );
    public MyRuntimeException (String message)
    {  super(message);  }
}
// throwing exception
throw new MyRuntimeException("My runtime message");
```

Can propagate out of method by itself
(no need to be thrown in method header)

Handling Exception

- All checked exceptions that may occur in try{…} must be caught

```
try {
    if (…) throw new MyException ( );
    if (…) throw new YourException ( );
}
catch (MyException e)       { … }
catch (YourException e)     { … }

// Exception not being thrown cannot be caught
catch (OtherException e)    { … }   Error!

// But parent exception can be caught after child
catch (Exception e)         { … }
```

- Catching multiple exceptions in the same block

```
try {
    if (…) throw new MyException ( );
    if (…) throw new YourException ( );
}
catch (MyException | YourException e)     { … }

// Parent and child are not allowed in the same block
catch (MyException | Exception e)         { … }    Error!
```

Method Throwing Exception

- void MyMethod (…) **throws** MyException { … }
  - An exception may be thrown out by this method
  - Any client calling this method must handle MyException
  - If the method throws multiple exceptions, its client must handle all of them
- Method (client) that calls MyMethod must either
  1) Catch & handle exception ➔ call MyMethod in try-catch
  2) Do not catch but let exception propagate
     - **throws** clause is also added to client's header
     - Anyone else calling this client must catch exception or let it propagate as well

```
public static void main(…)  throws XXException {
    int x = 10;
    if (…) {
        methodA( );
        x = x * 2;
    }
    x = x + 20;
}
```

JVM

Exception should be handled somewhere in the program. If it propagates to JVM, JVM will only show exception messages & end the program

```
public static void methodA( ) throws XXException {
    double a = 100.5;
    for (…) {
        methodB( );
        a = a * 2;
    }
    a = a + 200;
}
```

```
public static void methodB( ) throws XXException {
    int b = 200;
    while (…) {
        if (…) {
            throw new XXException(…);
            b = b * 3;
        }
        b = b + 300;
    }
    b = b / 2;
}
```

Overloading methods

- May throw different exceptions

```
void MyMethod (int k)       throws  MyException, YourException
void MyMethod (float k)     throws  MyException
void MyMethod (String s)    throws  OtherException
```

Overriding method in parent-child relationship

- General idea = *"successor is better"*
- Method throwing exception cannot override the one not throwing exception (child must not be worse than parent)
- Method throwing different exceptions cannot override each other (child should resolve parent's flaw & not add even more flaw)

Continuation

- There are 3 exit conditions for try-catch
  - Normal condition : exit at the end of try{…}
  - Exception condition : exit at the end of catch{…}
    - After the exception is caught and handled, the whole try-catch is said to be completed
    - If the exception is not caught but propagates to outer blocks, try-catch is also said to be completed
  - Exit on break, continue, return, or System.exit(*int*)
- In all conditions except System.exit(*int*), finally{…} is executed prior to the execution of next instruction
  - System.exit(*int*) immediately terminates the program

Try-with-resources

- Old Java
  - Resources should be closed or released in finally{…}
  - But doing it manually is cumbersome because the resources must be seen in both try{…} and finally{…}

```
Scanner scan = null;
try {
    scan = new Scanner( new File("input.txt") );
    … use scan …
}
catch(FileNotFoundException e)   { … }
finally {
    if (scan != null)  scan.close( );
    … other stuff that must always be done …
}
```

- Since Java 7, we can use **try-with-resources**
  - Resources that implement AutoCloseable (e.g. Scanner, PrintWriter, InputStream, OutputStream) can be declared

try (resources) {…}

  - They are local resources used only in try{…}
  - Automatically closed in reverse order of the declaration, once the execution leaves try{…}

```
try (
    Scanner scan = new Scanner( new File("input.txt") );
    PrintWriter write = new PrintWriter( new File("output.txt") );
) {
    … use scan & write …
}
catch(FileNotFoundException e)   { … }
finally {
    … other stuff that must always be done …
}
```