13 Review

Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

December 4, 2019

Outline

Visibility Modifiers

- Java provides several modifiers that control access to data fields and methods.
- Modifier `public` makes methods, and data fields accessible from any class.
- Modifier `private` makes methods and data fields accessible only from within its own class.
- Modifier `protected` makes methods and data fields accessible only from within its own class and its subclasses.
- When a subclass overrides a `protected` method, it may increase the visibility to `public`.

Data Field Encapsulation

- Data fields of an object describe the *state* of the object.
- To prevent the outside from modifying the data fields, the fields should be declared `private`. This is known as data field *encapsulation*.
- Encapsulation also makes data easy to maintain.
- We get and set data fields via public methods (called *getters* and *setters*).

```
public PropertyType getProperty() { ... }
public boolean isBooleanProperty() { ... }
public void setProperty(PropertyType propertyValue) { ... }
```

Constructors

- A *constructor* is a special kind of method designed to perform *initializing* actions, such as initializing the data fields of objects.
- A constructor operates on a newly created instance, invoked by "new".
- One constructor can invoke other constructors of the same class using "this" *before* any other statements.

```
class Circle {
    public Circle(double x, double y, double radius) {
        this.x = x; this.y = y; this.radius = radius;
    }
    public Circle(double radius) { this(0.0, 0.0, radius); }
    ...
}
```

The "this" Instance and Static Fields

- Within a method, "`this`" refers to the instance which the method is operating on.
- Within a method, if a name is not declared as a local variable or a parameter, it is prefixed with "`this.`" by default.
- A local variable or a parameter hides the field with the same name, to access the member, "`this.`" must be specified explicitly.
- When invoking *myCircle*.*contains*(1.0,1.0), the `this.`*x* in method *contains* refers to *myCircle*.*x*.
  When invoking `new` *Circle*(0.0,0.0,3.0), the `this.`*radius* in the constructor refers to the field *radius* in a newly created instance.
- A *static data field* does not belong to any instance. A static field is accessed via its class name *ClassName*.*staticField*, or accessed directly in a *static method* of the same class.

Immutable Objects and Classes

- If the contents of an object *cannot be changed* once the object is created, the object is called an *immutable* object and its class is called an immutable class.
- The *Circle* class in the example is immutable because *x*, *y* and *radius* are all private and there are no set methods to change them.

```
public class Circle {
    private double x, y, radius;
    public Circle(double x, double y, double radius) {
        this.x = x; this.y = y; this.radius = radius;
    }
    public double getArea() { return Math.PI*radius*radius; }
    public Circle scale(double f) { return new Circle(x, y, f*radius); }
}
```

- It is important that if a method is to update an immutable object, it must create and return a new object to reflect the update.

Declaring a Subclass

- The `extends` keyword is used to declare a subclass.

  `class` *Circle* `extends` *Shape* `{ ... }`

  where, *Circle* is the subclass, and *Shape* is the superclass of *Circle*.
- A subclass can have only one superclass in Java. This is called the single inheritance model.
- However, multiple subclasses can share one superclass.
- In a subclass, all the methods and attributes from the superclass are inherited. However, whether a particular member can be seen follows the visibility specification.
- For example, you can apply a `public` method of *Shape* to an object of *Circle*, but you cannot use the `private` field *color* directly outside the definition of *Shape*.

  ```
  Circle c = new Circle(); System.out.println(c.getColor());
  ```

Constructors of Superclasses

- Constructors of the superclass are *not* inherited. (Why not?)
- A constructor is used to construct an instance of a class. The construction must be complete.
- Although a (complete) object of *Circle* can be regarded as an object of *Shape*, the *Shape* class does not know how to make a *Circle*.
- Constructors of the superclass can be invoked in constructors of a subclass, to initialize the superclass portion, using the "`super`" keyword. Like `this(...)` calls, `super(...)` calls in constructors must appear in front of other statements.

```
Circle() { super("red", false); radius = 1.0; }
```

- If the keyword `super` is not explicitly used, and no other constructor is called via the keyword `this`, the default constructor of the superclass is automatically invoked.

```
Circle() { radius = 1.0; } ⟹ { super(); radius = 1.0; }
```

- A constructor of the superclass is *always* invoked, either explicitly or implicitly.

Constructor Chaining

```
public class Faculty extends Employee {
    public Faculty() { System.out.println("(4) Faculty()")); }
}
public class Employee extends Person {
    public Employee() {
        this("(2) Employee(String s)");
        System.out.println("(3) Employee()");
    }
    public Employee(String s) { System.out.println(s); }
}
public class Person {
    public Person() { System.out.println("(1) Person()"); }
}
```

A `new Faculty()` should print (1) — (4).

Constructor Chaining

```
public class Faculty extends Employee {
    public Faculty() { System.out.println("(4) Faculty()")); }
                   super();
}
public class Employee extends Person {
    public Employee() {
        this("(2) Employee(String s)");
        System.out.println("(3) Employee()");
    }
    public Employee(String s) { System.out.println(s); }
                            super();
}
public class Person {
    public Person() { System.out.println("(1) Person()"); }
}
```

A $\boxed{\texttt{new } \mathit{Faculty}()}$ should print (1) — (4).

Constructor Chaining

```
public class Faculty extends Employee {
    public Faculty() { System.out.println("(4) Faculty()")); }
}
                        super();
public class Employee extends Person {
    public Employee() {
        this("(2) Employee(String s)");
        System.out.println("(3) Employee()");
    }
    public Employee(String s) { System.out.println(s); }
}
                                   super();
public class Person {
    public Person() { System.out.println("(1) Person()"); }
}
```

A `new Faculty()` should print (1) — (4).

Overriding Methods of Superclasses

- A subclass inherits methods from the superclass.
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method *overriding*.

```
public class Circle extends Shape {
...
    @Override
    public String toString() {
        return super.toString() + "\nradius is " + radius;
    }
}
```

- A method can be overridden only if it is visible. Thus a `private` method cannot be overridden.
- Always use the `@Override` annotation to check overriding.

Polymorphism and Dynamic Binding

- An object of a *subclass* can be *used* wherever an object of the *superclass* is *required*. Thus a reference to a superclass object may refer an object of a subclass. This feature is known as polymorphism.
- For a *Shape x*, *x* may refer to an object of either *Shape*, *Circle*, *Rectangle* or *Triangle*, each of the classes may have their own implementation of method *toString*, due to method overriding.
- Which implementation is used will be determined dynamically, depending on *the actual class of the object* pointed to by *x* at *runtime*. This capability is known as *dynamic binding*.
- Methods are selected by instances. We often call non-static methods *instance methods*.
- As a consequence, `static` methods cannot be overridden, because they do not belong to instances, they belong to classes.

Abstract Methods

- Abstract methods capture the function, not the implementation.
- Abstract methods are placeholders that are meant to be overridden. Thus, we don't have `private` or `static` abstract methods.
- Abstract methods allow us to write code that makes use of a function without knowing the implementation, this helps to partially specify a *framework*.

```java
public abstract class Shape { ...
    public abstract double getArea();
    public static double getTotalArea (Shape[] ss) {
        double ta = 0.0;
        for ( Shape s : ss ) ta += s.getArea();
        return ta;
    }
}
```

Abstract Classes

- An abstract class is a class that is declared `abstract`.

  `public abstract class` *Shape* `...`

- An abstract class may or may *not* include abstract methods.
- Abstract classes cannot have instances of their own.
- Abstract classes can define constructors, which are invoked in the constructors of their subclasses.
- Abstract classes can be subclassed, and they are designed to be used as superclasses.
- An abstract method cannot be contained in a non-abstract class, directly or indirectly by inheritance.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared abstract.
- An abstract subclass can declare an abstract method to override a concrete method from its superclass.

Interfaces

- An interface is a class-like construct that contains *only* constants and abstract methods.
- To distinguish an interface from a class, Java uses the `interface` keyword to declare an interface.

```
public interface PenWidth {
    void setPenWidth(int w);
    int getPenWidth();
}
```

- In an interface, all methods are `public abstract`. Java allows these modifiers to be omitted.
- As with an abstract class, an interface cannot have instances of its own.
- Like an abstract class, an interface can be used as a data type for a variable, as the result of casting, and so on.

Interfaces

- An interface is a class-like construct that contains *only* constants and abstract methods.
- To distinguish an interface from a class, Java uses the `interface` keyword to declare an interface.

```
public interface PenWidth {
    public abstract void setPenWidth(int w);
    public abstract int getPenWidth();
}
```

- In an interface, all methods are `public abstract`. Java allows these modifiers to be omitted.
- As with an abstract class, an interface cannot have instances of its own.
- Like an abstract class, an interface can be used as a data type for a variable, as the result of casting, and so on.

Declaring Classes to Implement Interfaces

- We cannot set the pen width of a *Circle*, because *Circle* does not implement *PenWidth*.
- We can declare a subclass of *Circle* — *OutlinedCircle* to implement the interface while retaining all the features in *Circle*.

```
public class OutlinedCircle extends Circle implements PenWidth {
    private int pw = PenWidth.THIN;
    @Override public void setPenWidth(int width) { pw = w; }
    @Override public int getPenWidth() { return pw; }
}
```

- An instance of *OutlinedCircle* is also an instance of both *Circle* and *PenWidth*.

Anonymous Classes

- An anonymous class is a *local* class without a name.
- An anonymous class is *defined* and *instantiated* in a single expression using the `new` operator.
- An anonymous class must be used in conjunction with an interface or an (abstract) superclass.

```
public interface UnaryOp { int op(int x); }
```

- Now, we can define an object that carries a function by using an anonymous class.

```
public static UnaryOp incBy(final int delta) {
    return new UnaryOp() {
        @Override public int op(int x) { return x + delta; }
    };
}
```

Parametric Classes and Interfaces

- Java allows *parametric* types, called generics, to write certain types as *type parameters*.
- A type parameter can be constrained by a superclass and multiple interfaces.

```
class Node<E> {
    private E elem; ...
}
class ShapeNode<S extends Shape & PenWidth> {
    private S shape; ...
}
```

- When we use the generic class, we substitute actual types for the type parameters to *instantiate* a new class.
- The actual element type used to instantiate a generic type must be a class type. For a primitive type, their boxed type must be used instead.
- For example, $List\langle Integer\rangle$ is correct, but $List\langle \texttt{int}\rangle$ is *not*.

Instantiations

Generic Class: $Lst\langle T\rangle$

```
class Lst<T>
    T head;
    Lst<T> tail;

    T getHead() ...
```

$T \mapsto \underline{String}$

Instantiation: $Lst\langle \underline{String}\rangle$

```
class Lst<String>
    String head;
    Lst<String> tail;

    String getHead() ...
```

$T \mapsto \underline{Circle}$

Instantiation: $Lst\langle \underline{Circle}\rangle$

```
class Lst<Circle>
    Circle head;
    Lst<Circle> tail;

    Circle getHead() ...
```

The *Comparable* Interface

- The *compareTo* method is the only method in the $Comparable\langle T\rangle$ interface.
- If you are writing a class with an obvious natural ordering, such as alphabetical order, numerical order, or chronological order, you should consider implementing the interface.

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

- The *compareTo* method compares `this` object with the specified object for order and returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Generic Methods: Using the *Comparable* Interface

The following generic method finds the minimum element of a, b and c.

```
static <T extends Comparable<T>> T min(T a, T b, T c) {
    if ( a.compareTo(b) <= 0 && a.compareTo(c) <= 0 )
        return a;
    else if ( b.compareTo(c) <= 0 )
        return b;
    else
        return c;
}
```

Covariance and Contravariance

- Parametric types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Shape> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Circle> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Covariance and Contravariance

- Parametric types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Circle> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Circle> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Covariance and Contravariance

- Parametric types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Circle> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Shape> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Bounded Wildcard Types

- Upper bound: $Node\langle ?$ `extends` $Shape\rangle$ — $Node\langle$ any subclass of $Shape\rangle$.

  ```
  Shape getFromNode(Node<? extends Shape> n) {return n.getElem();}
  ```

- Lower bound: $Node\langle ?$ `super` $Circle\rangle$ — $Node\langle$ any superclass of $Circle\rangle$.

  ```
  void setToNode(Node<? super Circle> n, Circle c) {n.setElem(c);}
  ```

- We can also use them together to write a read-use-write example:

```
void readUseWrite(Node<? super Rectangle> dest,
                  Node<? extends Rectangle> src) {
    Rectangle r = src.getElem();
    System.out.println(r.getWidth()+","+r.getHeight());
    dest.setElem(r);
}
```

Iterables and Iterators

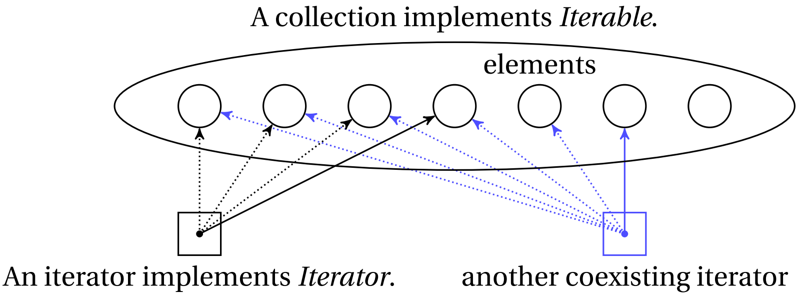
- A common operation of a collection is to enumerate its elements, such as to print all the elements in a linked list.
- In Java (and many other programming languages), such an operation is called *iteration*, which is closely related to the loop statement.
- If a collection is to support Java's standard iteration mechanism, it must implement the *Iterable* interface, in which it returns an instance of the *Iterator* interface (declared in *java.util*).

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

Iterables and Iterators (2)

- Method `boolean` *hasNext*() returns true if the iteration has more elements.
- Method *E next*() returns the next element in the iteration.



Implementing Iterables

An *Iterable* can return an instance of an anonymous class which implements the *Iterator*.

```
public class ZipStr implements Iterable<String> {
    private Iterable<String> a, b;
    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private Iterator<String> ia = a.iterator();
            private Iterator<String> ib = b.iterator();
            public boolean hasNext() { return ia.hasNext() && ib.hasNext(); }
            public String next() { return ia.next()+","+ib.next(); }
        };
    }
}
```

Java Collections Framework

- A collection is a container object that represents a group of objects, often referred to as *elements*.
- Some collections allow duplicate elements and others do not.
- Some collections are ordered and others are unordered.
- A map (also *associative array*) represents a group of objects, each of which is associated with a key. You can get the object from a map using a key, and you have to use a key to put the object into the map.
- The Java Collections Framework supports three types of collections, named *sets*, *lists*, and *maps*, grouped in the $java.util$ package.

The $List\langle E \rangle$ Interface

A list can not only store duplicate elements, but can also allow the user to specify *where* the element is stored. The user can access the element by *index*.

- `boolean` *add*(*E e*) — appends the specified element to the end of `this` list.
- `void` *add*(`int` *i*, *E e*) — inserts the specified element at the specified position in `this` list.
- *E get*(`int` *i*) — returns the element at the specified position in `this` list.
- *E set*(`int` *i*, *E e*) — replaces the element at the specified position in `this` list with the specified element.
- `int` *indexOf*(*Object o*) — returns the index of the first occurrence of the specified element in `this` list, or -1 if the element is not found.
- `int` *lastIndexOf*(*Object o*) — returns the index of the last occurrence of the specified element in `this` list, or -1 if the element is not found.

$ArrayList\langle E\rangle$ and $LinkedList\langle E\rangle$ are two implementation classes of $List\langle E\rangle$.

The *AbstractList* Class

This class provides a partial implementation of the *List* interface.

- An implementation backed by a "random access" data store (such as an array) can be based on *AbstractList*.
- To implement an unmodifiable list, we need to override the *get*(*i*) and *size*() methods.

```
class Cat<T> extends AbstractList<T> {
    private T[] a, b;
    public Cat(T[] a, T[] b) { this.a = a; this.b = b; }
    public T get(int i) { return i < a.length ? a[i] : b[i-a.length]; }
    public int size() { return a.length+b.length; }
}
```

- To implement a modifiable list, we must also override the *set*(*i*, *e*), *add*(*i*, *e*) and *remove*(*i*) methods.

Declaring and Throwing Exceptions

Java's exception-handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception.

- Every method must state the types of checked exceptions it might throw. This is known as declaring exceptions.

```
public void myMethod() throws IOException, MyException { ... }
```

- A program that detects an error can create an instance of an appropriate exception class and throw it. This is known as throwing an exception.

```
MyException ex = new MyException("Wrong Case");
throw ex;
```

Or,

```
throw new MyException("Wrong Case");
```

Catching Exceptions

- When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

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

- Only the `catch` block that *first* matches the exception type is entered. If no exceptions arise during the execution of the `try` block, all the `catch` blocks are skipped.

Exception Inheritance

- Various exception classes can be derived from a common superclass.
- If a catch block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.
- The order in which exceptions are specified in `catch` blocks is important.
- A compilation error will result if a `catch` block for a superclass type appears before a `catch` block for a subclass type.

```
try {
   ...
} catch ( Exception ex ) {
   ...
} catch ( RuntimeException ex ) {
   ...
}
```

```
try {
   ...
} catch ( RuntimeException ex ) {
   ...
} catch ( Exception ex ) {
   ...
}
```

Functional Interfaces

- In Java, a callback function is a method that belongs to an object. By passing and returning this object, the method can be carried with.
- For a standalone callback function, the minimal type of such a carrying object is an interface with only one abstract method.

```
interface Comparator<T> {
    int compare(T x, T y);
}

interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

- An interface with only one abstract method is thus called a *functional interface*.

Lambda Expressions

- With Java 8, a functional interface can be implemented by a lambda expression, which is a stateless object of an anonymous class.

  $$sort(a,\ (x,\ y)\ \texttt{->}\ Double.compare(y.mark,\ x.mark));$$

  $$exitButton.addActionListener(e\ \texttt{->}\ \{\ frame.dispose();\ \});$$

- A lambda expression has the following form,

  (parameter list) -> expression or { statements }

  A lambda expression defines an anonymous method.
- The name of the method and the types of the parameters and return value are inferred from the functional interface which the lambda expression implements.
- More important, usually, a callback function does not need to have a state.
- A lambda expression is an *object* instance of a functional interface.

Java *Stream* API — *reduce*

- The *reduce*() method can reduce the elements of a stream to a single value.

```
String reduced = stream.reduce((acc, item) -> acc + " " + item).get();
```

- This *reduce*() method takes a BinaryOperator as parameter and returns an *Optional* In case the stream contains no elements, the *Optional*.*get*() returns `null`.
- There is another *reduce*() method which takes two parameters. It takes an initial value for the accumulated value, and then a *BinaryOperator*.

```
String reduced2 = stream.reduce("", (acc, item) -> acc + " " + item);
```

- A stream can be reduced in parallel internally into several accumulators and then combined together for the final result.
- If combining two accumulations is different from accumulating a single element, we need to specify another binary operator for the combiner.

```
String reduced3 = stream.reduce("", (acc, item) -> acc + " [" + item +"]",
                                (acc1, acc2) -> acc1 + " " + acc2);
```

Functional Sets

- A functional set is a function that returns `true` or `false` to indicate whether it contains a certain element.

  `public interface` $FunSet\langle T\rangle$ `{ boolean` $contains(T\ x)$`; }`

- The empty set can be represented as a lambda expression: x `-> false`.
- To add an element x to a set s ($s \cup \{x\}$), we can write

  y `->` $x.equals(y)$ `||` $s.contains(y)$

- To remove an element x from a set s ($s \setminus \{x\}$), we can write

  y `-> !`$x.equals(y)$ `&&` $s.contains(y)$

- To make a union of two sets s and t ($s \cup t$), we can write

  y `->` $s.contains(y)$ `||` $t.contains(y)$

- To make an intersection of two sets s and t ($s \cap t$), we can write

  y `->` $s.contains(y)$ `&&` $t.contains(y)$

The *Thread* Class

- Since the *Thread* class implements *Runnable*, when a task is to be executed by only one thread, you could declare a class that extends *Thread* and overrides the *run* method.
- `void` *start*() — causes `this` thread to begin execution.
- `boolean` *isAlive*() — tests if `this` thread is alive. A thread is alive when it is running in the *run* method of the task.
- `void` *join*() — waits for `this` thread to die.
- `static void` *sleep*(`long` *millis*) — causes the *current thread* to sleep for the specified number of milliseconds, approximately.
- `static void` *yield*() — gives a hint to the scheduler that the current thread is willing to yield its current use of a processor.
- If, in the running of *threadA*, a call *threadB*.*join*() is made, then *threadA* is the current thread for the entire call, and *threadB* is the `this` thread for method *join*. That is, *threadA* waits for *threadB* to die.

The synchronized Keyword

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- Certain sequence of actions on an object cannot be interleaved with other actions.
- It is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the *critical region*.
- A `synchronized` method acquires a lock (on `this` object, or the class) before it executes.

```
public synchronized void deposit(double amount) {
    this.balance = this.balance+amount;
}
```

- A synchronized statement can be used to acquire a lock on any object, when executing a block of statements.

```
synchronized ( obj ) { obj.use(); }
```

Avoiding Deadlocks

- Two or more threads may need to acquire the locks on several shared objects.
- This could cause a deadlock, in which each thread has the lock on one of the objects and is waiting for the lock on the other object.

Thread 1:

```
synchronized ( a ) {
    ...
    synchronized ( b ) { ★
        ...
    }
}
```

Thread 2:

```
synchronized ( b ) {
    ...
    synchronized ( a ) { ★
        ...
    }
}
```

- Deadlock can be avoided by using a simple technique known as *resource ordering*.
- You assign an order on all the locks and ensure that each thread acquires the locks in that order.