西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object Oriented Programming

Chapter 6
Interfaces, Lambda Expressions, and Inner Classes

Dr. Helei Cui

18 Apr 2021

Slides partially adapted from lecture notes by Cay Horstmann

Contents

6.1.1 The Interface Concept

- What is an interface in Java?
  - Interface is not a class, but a set of requirements for classes.
- Class can choose to conform to one or more interfaces.
  - "If your class conforms to a particular interface, then I'll perform the service."
  - **Example:** `Arrays.sort` sorts an array if the element class conforms to the `Comparable` interface.

```
public interface Comparable {
   int compareTo(Object other); // automatically public
}
```

  - *This means that any class that implements the* `Comparable` *interface is required to have a* `compareTo` *method, and the method must take an* `Object` *parameter and return an integer.*
- **All methods of an interface are automatically `public`.**

6.1.1 The Interface Concept

- Interfaces can define constants, but never have instance fields.
  - An interface = An abstract class with no instance fields
- Two steps to make a class implement an interface:
  1. Declare that your class intends to implement the given interface.
  2. Supply definitions for all methods in the interface.

```java
public class Employee implements Comparable {
   public int compareTo(Object otherObject) {
      Employee other = (Employee) otherObject;
      return Double.compare(salary, other.salary);
   }
   . . .
}
```

Caution

```
public interface Comparable {
   int compareTo(Object other); // automatically public
}
```

- In the interface declaration, the compareTo method was not declared public because all methods in an interface are **automatically public**.

```
public class Employee implements Comparable {
   public int compareTo(Object otherObject) {
      Employee other = (Employee) otherObject;
      return Double.compare(salary, other.salary);
   }
}
```

- However, when implementing the interface, you must declare the method as public.
  - Otherwise, the compiler assumes that the method has package access - the default for a class. The compiler then complains that you're trying to supply a more restrictive access privilege.

6.1.1 The Interface Concept

- Better to supply a type parameter for the generic `Comparable` interface:

```
class Employee implements Comparable<Employee> {
    public int compareTo(Employee other) {
        return Double.compare(salary, other.salary);
    }
    . . .
}
```

- The `compareTo` method returns:
  - a negative if the first argument is less than the second argument;
  - 0 if they are equal;
  - a positive value otherwise.

A Problem?

Why can't the Employee class simply provide a compareTo method without implementing the Comparable interface?

- **Reason: the Java programming language is strongly typed.** When making a method call, the compiler needs to be able to check that the method actually exists.
  - Somewhere in the sort method will be statements like this:

```
if (a[i].compareTo(a[j]) > 0) {
   // rearrange a[i] and a[j]
   . . .
}
```

  - The compiler must know that a[i] has a compareTo method. If a is an array of Comparable objects, then the existence of the method is assured because every class that implements the Comparable interface must supply the method.

6.1.2 Properties of Interfaces

- **Interfaces are not classes. You can't use the new operator to instantiate an interface:**

```
x = new Comparable(. . .); // Error
```

- **You can have variables of interface type:**

```
Comparable x;                        // OK
```

  - The variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

  - Use `instanceof` to check whether an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

6.1.2 Properties of Interfaces

- **An interface can extend another:**

```
public interface Moveable {
   void move(double x, double y);
}

public interface Powered extends Moveable {
   double milesPerGallon();
}
```

- **An interface can have constants:**

```
public interface Powered extends Moveable {
    double milesPerGallon();
    double SPEED_LIMIT = 95; // a public static final constant
}
```

- **A class can implement multiple interfaces:**

```
class Employee implements Comparable, Moveable { ... }
```

6.1.3 Interfaces and Abstract Classes

- **Why not make `Comparable` into an abstract class?**

```
abstract class Comparable { // why not?
   public abstract int compareTo(Object other);
}
```

  - Then `Employee` would simply extend it:

```
class Employee extends Comparable { // why not?
   public int compareTo(Object other) { . . . }
}
```

- **A major problem: A class can only extend a single class.**

```
class Employee extends Person, Comparable // Error
```

  - But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

6.1.4 Static and Private Methods

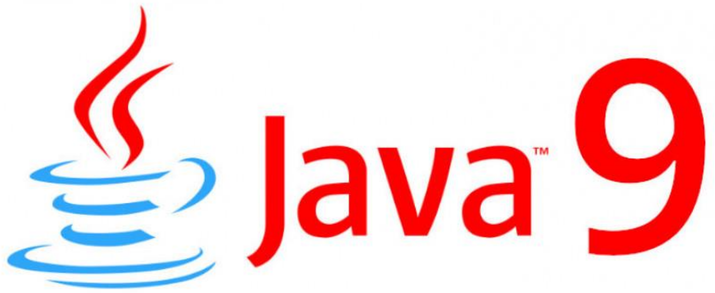
- Originally disallowed since it wasn't in the spirit of interfaces as abstract specifications.
  - As of Java 8, you are allowed to add static methods to interfaces.
- You can find many pairs of interface/companion class in the Java API: `Collection/Collections`, `Path/Paths`.
  - `Paths` has a factory method get to make a `Path` object.
    - *E.g., `Paths.get("jdk-11", "conf", "security")`.*
  - It would be better solved with a `static` method in the `Path` interface:

```
public interface Path {
    public static Path of(URI uri) { . . . }
    public static Path of(String first, String... more) { . . . }
    . . .
}
```

Similarly, when implementing your own interfaces, there is no need to provide a separate companion class for utility methods.

6.1.4 Static and Private Methods

- **As of Java 9, methods in an interface can be `private`.**
  - A private method can be `static` or an instance method.
  - Since `private` methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.



6.1.5 Default Methods

- **You can supply a `default` implementation for any interface method:**

```
public interface Comparable<T> {
    default int compareTo(T other) { return 0; }
        // by default, all elements are the same
}
```

  - Not very useful since every implementation of `Comparable` would override this method.
  - But sometimes, default methods can be useful.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    . . .
}
```

6.1.5 Default Methods

- **A default method can call an abstract method:**

```
public interface Collection {
     int size(); // an abstract method
     default boolean isEmpty() { return size() == 0; }
     . . .
}
```

- **An important use for default methods is *interface evolution*.**

```
public class Bag implements Collection
```

  - Consider in a later version, a new method is added to the interface.
  - If the method is not a default method, then the Bag class would no longer compile since it doesn't implement the new method.
    - ***Adding a nondefault method to an interface is not source-compatible.***
  - If you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method.
    - ***Adding a method to an interface is binary compatible.***
    - But if a program calls the new method on a Bag instance, an error occurs.
  - **Making the method a default method solves both problems.**

6.1.6 Resolving Default Method Conflicts

- **What happens if the exact same method is defined as a default method in one interface and then again as a method of a superclass or another interface?**

- Two simple rules in Java:
  1. **Interfaces clash.** If an interface provides a default method and another interface provides the same one (default or not), you must resolve the conflict **by overriding that method**.

  2. **Superclasses win.** If a superclass provides a concrete method, default methods with the same name and parameter types are **simply ignored**.

The “Interfaces Clash” Rule

```
interface Person {
    default String getName() { return ""; };
}
interface Named {
    default String getName() { return getClass().getName() +
     "_" + hashCode();
}
```

- **What happens if a class implements both interfaces?**
  - You need to implement the getName method.
  - You can call one of the two conflicting methods as follows:

```
class Student implements Person, Named {
    public String getName() { return Person.super.getName(); }
    . . .
}
```

The “Superclasses Win” Rule

- Now, assume that `Person` is a superclass and `Named` is an interface:

```
class Student extends Person implements Named { . . . }
```

- **What happens if the extended class and the implemented interface have the same methods?**
  - Only the superclass method matters, and any default method from the interface is simply ignored.
    - The default method `getName` in the `Named` interface will be ignored.
  - This rule ensures compatibility with Java 7:
    - If you add a default method to an interface, it has no impact on existing code.

Private Interface Methods

- As of Java 9:
  - Interfaces can have concrete `private` and `private static` methods.
  - Any interface method is abstract, `default`, `static`, `private`, or `private static`.
  - Private methods can only be called from `default` and `static` methods of the same interface.
  - Potentially useful for factoring out common code.

6.1.7 Interfaces and Callbacks

- **Callback:** Action that should happen when an event occurs.
  - E.g., `Timer` makes callback whenever a time interval has elapsed.
  - Give the timer an object of a class that implements this interface:

```
public interface ActionListener {
    void actionPerformed(ActionEvent event);
}
```

    - The timer calls the `actionPerformed` method when the time interval has expired.
  - You can define a class that implements the `ActionListener` interface:

```
class TimePrinter implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

6.1.7 Interfaces and Callbacks

- Construct and install the object:

```
var listener = new TimePrinter();
Timer t = new Timer(1000, listener);
t.start();
```

- Every second, a message is shown, followed by a beep.

```
At the tone, the time is 2017-12-16T05:01:49.550Z
```

- Try the Listing 6.3 timer/TimerTest.java!

6.1.8 The Comparator Interface

- You saw how `Arrays.sort` sorts an array of `Comparable` objects.
    - E.g., you can sort an array of strings since the `String` class implements `Comparable<String>`.
- **What if you want to sort the objects in a different way?**
- **What if the objects belong to a class that doesn't implement Comparable?**
    - To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a comparator - an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T> {
   int compare(T first, T second);
}
```

- https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Arrays.html#sort(T%5B%5D,java.util.Comparator)

6.1.8 The Comparator Interface

- This comparator compares strings by length:

```java
class LengthComparator implements Comparator<String> {
   public int compare(String first, String second) {
      return first.length() - second.length();
   }
}
```

- Do the comparison:

```java
var comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) . . .
```

- Pass an instance to `Arrays.sort`:

```java
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

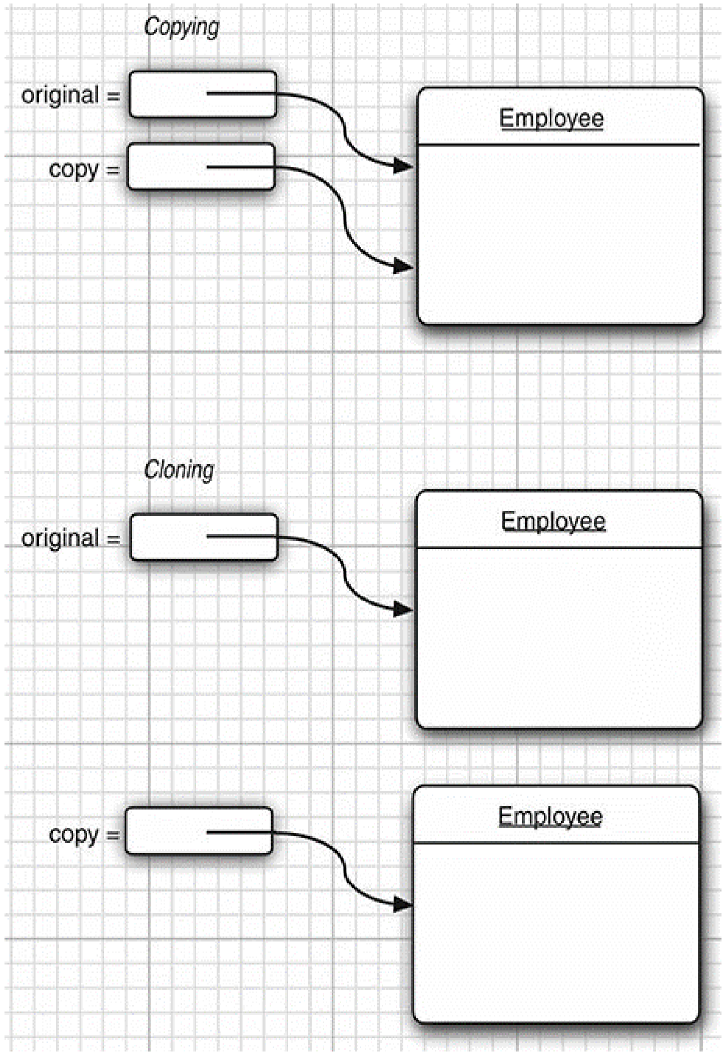
6.1.9 Object Cloning

- Recall what happens when you make a copy of an object variable:

```
var original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10);  // also changed original
```

- The `Cloneable` interface indicates that a class provides a safe clone method.
    - If `Employee` is cloneable, then you can call

```
Employee copy = original.clone();
copy.raiseSalary(10);  // original unchanged
```

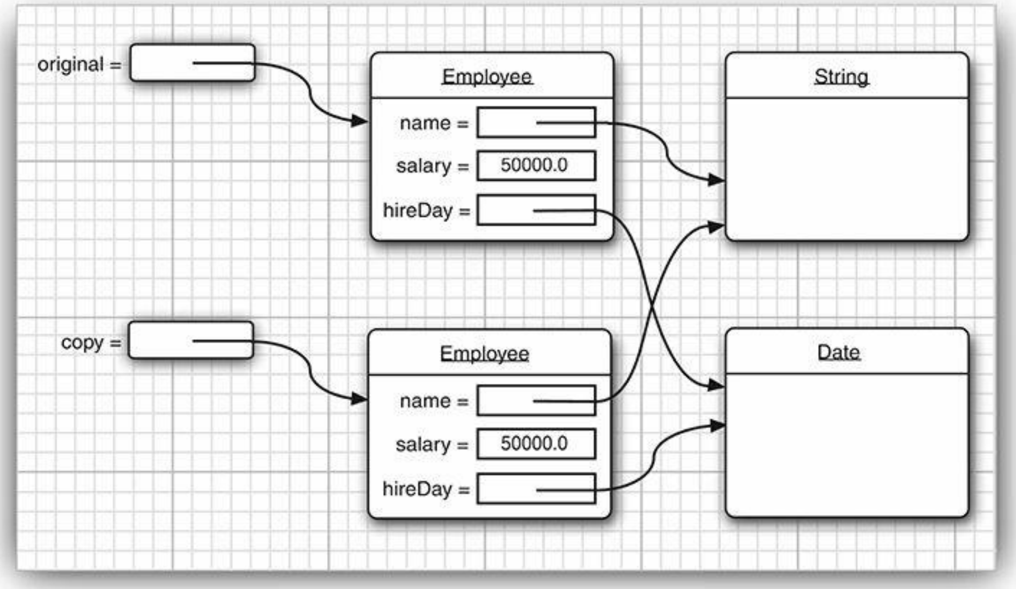


6.1.9 Object Cloning

- Cloneable is an interface without methods:

```
public interface Cloneable {}
```

- The clone method is a protected method of Object.
  - It means that your code cannot simply call it.
  - The method is protected because it is tricky to implement correctly.
  - **Object.clone makes a "shallow" copy: a new object with the same fields.**
  - **That is bad if one of the fields is a reference to a mutable object:**



6.1.9 Object Cloning

- **For every class, you need to decide whether**
  1. The default `clone` method is good enough;
  2. The default `clone` method can be patched up by calling `clone` on the mutable subobjects; or
  3. `clone` should not be attempted.

- **The third option is actually the default. To choose either the first or the second option, a class must**
  1. Implement the `Cloneable` interface; and
  2. Redefine the `clone` method with the `public` access modifier.

6.1.9 Object Cloning

- **You must implement a deep copy and clone any mutable fields:**

```
class Employee implements Cloneable {
    . . .
    public Employee clone() throws CloneNotSupportedException {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();
        return cloned;
    }
}
```

- You can catch the `CloneNotSupportedException` in a `final` class. Otherwise, it's better to leave the throws specifier in place.
- Less than 5% of the classes in the Java API are cloneable.

When to use Abstract Class?

- Consider using abstract classes if any of these statements apply to your situation:
  - In the java application, there are some related classes that need to share some lines of code then you can put these lines of code within the abstract class and this abstract class should be extended by all these related classes.
  - You can define the non-static or non-final field(s) in the abstract class so that via a method you can access and modify the state of the Object to which they belong.
  - You can expect that the classes that extend an abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

When to use Interface?

- Consider using interfaces if any of these statements apply to your situation:
  - It is a total abstraction, all methods declared within an interface must be implemented by the class(es) that implements this interface.
  - A class can implement more than one interface. This can be viewed as “multiple inheritances”.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.

Contents

6.2.1 Why Lambdas?

- **A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.**
- If you want to sort strings by length instead of the default dictionary order, you can pass a `Comparator` object to the `sort` method:

```
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
. . .
Arrays.sort(strings, new LengthComparator());
```

  - A block of code was passed to a `sort` method. That code block was called at some later time.

6.2.2 The Syntax of Lambda Expressions

- We pass code that checks whether one string is shorter than another. We compute "first.length() - second.length()".
  - The "first" and the "second" are both strings.
  - Now you can define your first lambda expression:

```
(String first, String second) -> first.length() - second.length()
```

- **Simplest form:** `(parameters) -> expression`

6.2.2 The Syntax of Lambda Expressions

- If the code doesn't fit in a single expression, use { } and a `return` statement:

```
(String first, String second) -> {
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

- If there are no parameters, you still supply parentheses:

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

- If parameter types can be inferred, you can omit them:

```
Comparator<String> comp
    = (first, second)      // same as (String first, String second)
        -> first.length() - second.length();
```

6.2.2 The Syntax of Lambda Expressions

- If a method has a single parameter with inferred type, you can even omit the parentheses:

```
ActionListener listener = event ->
     System.out.println("The time is "
     + Instant.ofEpochMilli(event.getWhen()));
     // instead of (event) -> . . . or (ActionEvent event) -> . . .
```

- You never specify the result type of a lambda expression. It is always inferred from context.

```
(String first, String second) -> first.length() - second.length()
```

  - A result of type `int` is expected.

6.2.3 Functional Interfaces

- Functional interface = Interface with a single abstract method (called functional method).
  - E.g., `ActionListener` and `Comparator`.
- Lambda expression can be used whenever a functional interface value is expected:

```
Arrays.sort(strings, new LengthComparator());   //functional interface
Arrays.sort(words,
    (first, second) -> first.length() - second.length());

var timer = new Timer(1000, new TimePrinter()); //functional interface
var timer = new Timer(1000, event -> {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    });
```

In fact, conversion to a functional interface is the only thing that you can do with a lambda expression in Java.

6.2.3 Functional Interfaces

- The `java.util.function` package defines generic functional interfaces.
- Example 1: `BiFunction<T, U, R>`, describes functions with parameter types T and U and return type R.

```
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
}

BiFunction<String, String, Integer> comp
    = (first, second) -> first.length() - second.length();
```

  - However, that does not help you with sorting. There is no `Arrays.sort` method that wants a `BiFunction`.
  - When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

6.2.3 Functional Interfaces

- Example 2: `Predicate<T>` represents a predicate (boolean-valued function) of one argument.

```
public interface Predicate<T> {
   boolean test(T t);
   // additional default and static methods
}
```

- `ArrayList` has a `removeIf` method that takes a `Predicate`.
  - It is specifically designed to pass a lambda expression.
  - E.g., this statement removes all `null` values from an array list:

```
list.removeIf(e -> e == null);
```

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/ArrayList.html#removeIf(java.util.function.Predicate)

6.2.3 Functional Interfaces

- Example 3: `Supplier<T>` represents a supplier of results.

```
public interface Supplier<T> {
    T get();
}
```

  - A supplier has no arguments and yields a value of type `T` when it is called. Suppliers are used for lazy evaluation.
  - E.g., consider the call

```
LocalDate hireDay = Objects.requireNonNullOrElse(day,
    new LocalDate(1970, 1, 1));
```

  - We expect that day is rarely `null`, so we only want to construct the default `LocalDate` when necessary.

```
LocalDate hireDay = Objects.requireNonNullOrElseGet(day,
    () -> new LocalDate(1970, 1, 1));
```

  - The `requireNonNullOrElseGet` method only calls the supplier when the value is needed.

6.2.4 Method References

- Consider a lambda expression that calls a single method:

```
var timer = new Timer(1000, event -> System.out.println(event));
```

  - It would be nicer if you could just pass the `println` method to the `Timer` constructor. Here is how you do that:

```
var timer = new Timer(1000, System.out::println);
```

- **"`System.out::println`" is a method reference.**
  - It directs the compiler to produce an instance of a functional interface, overriding the single abstract method of the interface to call the given method.
  - In this example, an `ActionListener` is produced whose `actionPerformed(ActionEvent e)` method calls `System.out.println(e)`.

Like a lambda expression, a method reference is not an object. It gives rise to an object when assigned to a variable whose type is a functional interface.

6.2.4 Method References

- As another example, suppose you want to sort strings regardless of letter case. You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

- **The :: operator separates the method name from the name of an object or class.** There are three variants:
    1. *object::instanceMethod*

```
System.out::println          // x -> System.out.println(x)
```

    2. *Class::instanceMethod*

```
String::compareToIgnoreCase //(x, y) -> x.compareToIgnoreCase(y)
```

    3. *Class::staticMethod*

```
Math::pow                    // (x, y) -> Math.pow(x, y)
```

Note

- A lambda expression can only be rewritten as a method reference if the body of the lambda expression calls a single method and doesn't do anything else.
  - Consider the lambda expression:

```
s -> s.length() == 0
```

    - There is a single method call. But there is also a comparison, so you can't use a method reference here.
- When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean.
  - Two versions of the `Math.max` method, which one gets picked depends on the method parameters of the functional interface.

“this” and “super” in Method References

- You can capture the this parameter in a method reference.

```
this::equals   // x -> this.equals(x)
```

- It is also valid to use super.

```
class Greeter {
    public void greet(ActionEvent event) {
        System.out.println("Hello, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
    }
}
class RepeatedGreeter extends Greeter {
    public void greet(ActionEvent event) {
        var timer = new Timer(1000, super::greet);
        timer.start();
    }
}
```

  - When the RepeatedGreeter.greet method starts, a Timer is constructed that executes the super::greet method on every timer tick.

6.2.5 Constructor References

- Constructor references are just like method references, **except that the name of the method is `new`.**
  - `Person::new` is a reference to a `Person` constructor.
  - Same as `s -> new Person(s)`.
  - The compiler uses overloading resolution to pick the correct constructor. E.g., turn list of names into list of `Person` objects:

```
ArrayList<String> names = . . .;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.collect(Collectors.toList());
```

  - The `map` method turns a stream of strings into a stream of `Person` objects.
    - It calls the `Person(String)` constructor for each list element.
    - If there are multiple `Person` constructors, the compiler picks the one with a `String` parameter because it infers from the context that the constructor is called with a string.

6.2.5 Constructor References

- Constructor references also work for arrays:
  - `int[]::new` is the same as the lambda expression `x->new int[x]`
  - Useful to overcome limitation of Java generics: illegal to call `new T[n]`
    - The expression `new T[n]` is an error since it would be erased to `new Object[n]`. That is a problem for library authors.
  - Suppose we want to have an array of `Person` objects. The `Stream` interface has a `toArray` method that returns an `Object` array:

```
Object[] people = stream.toArray();
```

- The user wants an array of references to `Person`, not references to `Object`. The stream library solves that problem with constructor references. Pass `Person[]::new` to the `toArray` method:

```
Person[] people = stream.toArray(Person[]::new);
```

- The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

6.2.6 Variable Scope

- A lambda expression can access variables from the enclosing scope:

```
public static void repeatMessage(String text, int delay) {
    ActionListener listener = event -> {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

- Consider a call:

```
repeatMessage("Hello", 1000); //prints Hello every 1,000 milliseconds
```

The code of the lambda expression may run long after the call to `repeatMessage` has returned, and the parameter variables are gone. How does the `text` variable stay around?

6.2.6 Variable Scope

- **A lambda expression has three ingredients:**
  1. A block of code;
  2. Parameters;
  3. Values for the *free* variables - that is, the variables that are not parameters and not defined inside the code.
- In above example, the lambda expression has one free variable, `text`.
  - The data structure representing the lambda expression must store the values for the free variables - in our case, the string "Hello". We say that such values have been captured by the lambda expression.

The technical term for a block of code together with the values of the free variables is a closure.

6.2.6 Variable Scope

- A lambda variable can only capture a variable whose value is unchanged:

```
public static void countDown(int start, int delay) {
    ActionListener listener = event -> {
            start--; // ERROR: Can't mutate captured variable
            System.out.println(start);
        };
    new Timer(delay, listener).start();
}
```

- Also, illegal if the variable changes outside the lambda expression:

```
public static void repeat(String text, int count) {
    for (int i = 1; i <= count; i++) {
        ActionListener listener = event -> {
                System.out.println(i + ": " + text);
                    // ERROR: Cannot refer to changing i
            };
        new Timer(1000, listener).start();
    }
}
```

The rule is that any captured variable in a lambda expression must be effectively final.

6.2.6 Variable Scope

- The body of a lambda expression has the same scope as a nested block.
  - The same rules for name conflicts and shadowing apply.
  - It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
Path first = Path.of("/usr/bin");
Comparator<String> comp
    = (first, second) -> first.length() - second.length();
    // ERROR: Variable first already defined
```

  - Inside a method, you can’t have two local variables with the same name, and therefore, you can’t introduce such variables in a lambda expression either.

6.2.6 Variable Scope

- When you use this keyword in a lambda expression, you refer to this parameter of the method that creates the lambda.

```
public class Application {
    public void init() {
        ActionListener listener = event -> {
            System.out.println(this.toString());
            . . .
        }
        . . .
    }
}
```

- The expression this.toString() calls the toString method of the Application object, not the ActionListener instance.
- There is nothing special about the use of this in a lambda expression.
- The scope of the lambda expression is nested inside the init method, and this has the same meaning anywhere in that method.

6.2.7 Processing Lambda Expressions

- **The point of using lambdas is *deferred execution*.**
- Reasons for executing code later:
  - Running the code in a separate thread;
  - Running the code multiple times;
  - Running the code at the right point in an algorithm (for example, the comparison operation in sorting);
  - Running the code when something happens (a button was clicked, data has arrived, and so on);
  - Running the code only when necessary;

6.2.7 Processing Lambda Expressions

- Example 1: repeat an action n times:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

- **To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface.**
  - In this example, use the `Runnable` interface (that runs an action without arguments or return value):

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

  - Note that the body of the lambda expression is executed when `action.run()` is called.

6.2.7 Processing Lambda Expressions

- Example 2: repeat an action n times and tell the action in which iteration it occurs.
  - For this, we can use the `IntConsumer` interface (that has a method with an `int` parameter and a `void` return):

```java
public interface IntConsumer {
    void accept(int value);
}
```

  - Here is the improved version of the `repeat` method:

```java
public static void repeat(int n, IntConsumer action) {
    for (int i = 0; i < n; i++) action.accept(i);
}
```

  - You can call it:

```java
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

6.2.8 More about Comparators

- Comparator interface has useful static methods for creating and composing comparators.
  - These methods are intended to be used with lambda expressions or method references.
- E.g., the static method comparing makes a comparator from a key extractor function:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

  - Chains comparators via thenComparing method:

```
Arrays.sort(people,
   Comparator.comparing(Person::getLastName)
      .thenComparing(Person::getFirstName));
```

  - If the key is a primitive type, use comparingInt or comparingDouble to avoid boxing:

```
Arrays.sort(people, Comparator.comparingInt(p ->
    p.getName().length()));
```

Contents

- 6.1 Interfaces
- 6.2 Lambda Expressions
- 6.3 Inner Classes

What is Inner Class?

- **An inner class is a class that is defined inside another class.**
- Two reasons why we need it:
  - Inner classes can be hidden from other classes in the same package.
  - Inner class methods can access the data from the scope in which they are defined - including the data that would otherwise be private.

- Notice:
  - Inner classes used to be very important for concisely implementing callbacks, but nowadays lambda expressions do a much better job.
  - Still, inner classes can be very useful for structuring your code.

6.3.1 Use of an Inner Class to Access Object State

- We refactor the `TimerTest` example and extract a `TalkingClock` class. A talking clock is constructed with two parameters:
  - the interval between announcements, and
  - a flag to turn beeps on or off.

```
public class TalkingClock {
    private int interval;
    private boolean beep;
    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }
    public class TimePrinter implements ActionListener {
        // an inner class
        . . .
    }
}
```

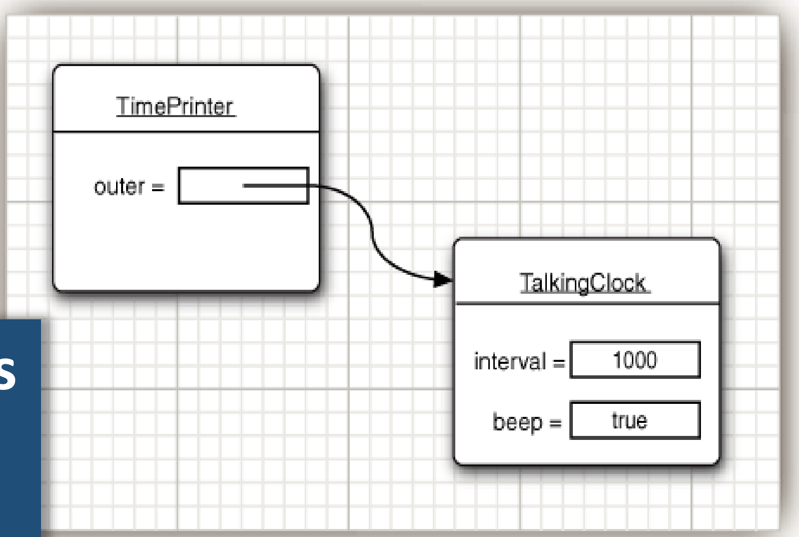
- The `TimePrinter` class is now located inside the `TalkingClock` class, but this does not mean that every `TalkingClock` has a `TimePrinter` instance field.
  - As you will see, the `TimePrinter` objects are constructed by methods of the `TalkingClock` class.

6.3.1 Use of an Inner Class to Access Object State

- Inner class implementation:

```
public class TimePrinter implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

- The TimePrinter class has no instance field or variable named beep.
    - Instead, beep refers to the field of the TalkingClock object that created this TimePrinter.



An inner class method gets to access both its own data fields and those of the outer object creating it.

6.3.1 Use of an Inner Class to Access Object State

- The reference to the outer object is called *outer*.

```
public void actionPerformed(ActionEvent event) {
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

- The outer class reference is set in the constructor.
    - The compiler modifies all inner class constructors, adding a parameter for the outer class reference.

```
public TimePrinter(TalkingClock clock) {
    // automatically generated code
    outer = clock;
}
```

6.3.1 Use of an Inner Class to Access Object State

- The “*outer*” is not a Java keyword, just used to illustrate the mechanism involved in an inner class.
  - When a `TimePrinter` object is constructed in the `start` method, the compiler passes the `this` reference to the current talking clock into the constructor:

```
var listener = new TimePrinter(this); // parameter automatically added
```

- Note:
  - We could have declared the `TimePrinter` class as `private`. Then only `TalkingClock` methods would be able to construct `TimePrinter` objects.
  - Only inner classes can be private. Regular classes always have either package or public access.

6.3.2 Special Syntax Rules for Inner Classes

- The reference to the outer class is `outerClass.this`:

```
public void actionPerformed(ActionEvent event) {
    . . .
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

- Use the syntax to write the inner object constructor.

```
outerObject.new InnerClass(construction parameters)
// Example: ActionListener listener = this.new TimePrinter();
```

- Any outer class object can construct an inner class object:

```
var jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

The name of a (non-private) inner class is `OuterClass.InnerClass` when it occurs outside the scope of the outer class.

6.3.3 Are Inner Classes Useful? Actually Necessary? Secure?

- Inner classes are translated into regular class files with $ (dollar signs) delimiting outer and inner class names, and the virtual machine does not have any special knowledge about them.
  - E.g., the `TimePrinter` class inside the `TalkingClock` class is translated to a class file `TalkingClock$TimePrinter.class`.

```
public class TalkingClock {
    private int interval;
    private boolean beep;
    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }
    public class TimePrinter implements ActionListener {
        // an inner class
        . . .
    }
}
```

6.3.3 Are Inner Classes Useful? Actually Necessary? Secure?

- Let's try to make `TimePrinter` a regular class, outside the `TalkingClock` class.
  - When constructing a `TimePrinter` object, we pass it the `this` reference of the object that is creating it.

```
class TalkingClock {
    public void start() {
        var listener = new TimePrinter(this);
        var timer = new Timer(interval, listener);
        timer.start();
    }
}
class TimePrinter implements ActionListener {
    private TalkingClock outer;
    public TimePrinter(TalkingClock clock) {
        outer = clock;
    }
}
```

6.3.3 Are Inner Classes Useful? Actually Necessary? Secure?

- Later, let’s see the `actionPerformed` method. It needs to access `outer.beep`.

```
if (outer.beep) . . . // ERROR
```

- **Reason:**
  - The inner class can access the private data of the outer class, but our external `TimePrinter` class cannot.
    - Thus, inner classes are genuinely more powerful than regular classes because they have more access privileges.

6.3.3 Are Inner Classes Useful? Actually Necessary? Secure?

- To summarize, if an inner class accesses a private data field, then it is possible to access that data field through other classes added to the package of the outer class, but to do so requires skill and determination.
- A programmer cannot accidentally obtain access but must intentionally build or modify a class file for that purpose.

6.3.4 Local Inner Classes

- If an inner class is only used in a method, you can define the class **locally** in a single method, called **local inner class**.
  - E.g., we can define `TimePrinter` class in a single method.

```
public void start() {
    class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.start();
}
```

  - Local classes are never public, private, or protected.
    - Scope is always restricted to the block in which they are declared.
  - **One advantage: they are completely hidden from the outside world.**
    - Not even other code in the `TalkingClock` class can access them.
    - No method except `start` has any knowledge of the `TimePrinter` class.

6.3.5 Accessing Variables from Outer Methods

- Local classes can access *effectively* final variables from the enclosing scope.
  - You can move the interval and beep parameters from the TalkingClock constructor to the start method.

```
public void start(int interval, boolean beep) {
     class TimePrinter implements ActionListener {
          public void actionPerformed(ActionEvent event) {
               System.out.println("At the tone, the time is "
               + Instant.ofEpochMilli(event.getWhen()));
               if (beep) Toolkit.getDefaultToolkit().beep();
          }
     }
     var listener = new TimePrinter();
     var timer = new Timer(interval, listener);
     timer.start();
}
```

  - Note that the TalkingClock class no longer needs to store a beep instance field. It simply refers to the beep parameter variable of the start method.

6.3.5 Accessing Variables from Outer Methods

- Consider the flow of control more closely:
  1. The `start` method is called.
  2. The object variable `listener` is initialized by a call to the constructor of the inner class `TimePrinter`.
  3. The `listener` reference is passed to the `Timer` constructor, the timer is started, and the `start` method exits. At this point, the `beep` parameter variable of the `start` method no longer exists.
  4. A second later, the `actionPerformed` method executes "`if (beep) . . .`"

6.3.6 Anonymous Inner Classes

- **If a local class is only instantiated once, it can be anonymous:**

```
public void start(int interval, boolean beep) {
    var listener = new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    var timer = new Timer(interval, listener);
    timer.start();
}
```

- This means:
    - Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces `{ }`.

6.3.6 Anonymous Inner Classes

- General syntax of anonymous inner class:

```
new SuperType(construction parameters) {
    // inner class methods and fields
}
```

  - SuperType can be an interface, such as `ActionListener`; then, the inner class implements that interface.
  - SuperType can also be a class; then, the inner class extends that class.
- An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name.
  - Instead, the construction parameters are given to the superclass constructor.
  - In particular, whenever an inner class implements an interface, it cannot have any construction parameters.

6.3.6 Anonymous Inner Classes

- If there is just one method, use a lambda expression.
  - E.g., the `start` method from the beginning of this section can be written much more concisely with a lambda expression like this:

```
public void start(int interval, boolean beep) {
    var timer = new Timer(interval, event -> {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    });
    timer.start();
}
```

6.3.7 Static Inner Classes

- **Static inner class = inner class without reference to creating object.**
- Useful for a private or scoped class that doesn't need to know the creating object.

```
class ArrayAlg {
   public static class Pair {
      public double first;
      public double second;
   }
   . . .
   public static Pair minmax(double[] values) {
      . . .
      return new Pair(min, max); // no creating object
   }
}
```

- Called as:

```
ArrayAlg.Pair p = ArrayAlg.minmax(data);
```

Recap

- 6.1 Interfaces
- 6.2 Lambda Expressions
- 6.3 Inner Classes