CS 210 Lecture Notes

Nicholas Kossiaris

20 Mar 2026 - present

Theory

Abstract data types (ADT) A specification of the behavior of a system without specifying the implementation.
Nothing: An abstract algorithm that cannot fulfill its calling an abstract method is reserved for data structures that, unlike subclasses implement only the constructing method is not a valid implementation.

Data structure A concrete realization of the behavior specified by an ADT.

Unspecified complexity Unspecified complexity means that a method is required to exist but the complexity is not specified.

Composition pattern Composition is a design pattern where a class contains other objects and delegates work to them. The phrase "has a" relationship.

Pure Pure is the opposite of composition.

Iterator pattern An iterator pattern is a design pattern that allows a client to traverse a collection without knowing the underlying data structure. Clients call `next()` and `hasNext()` to receive elements one at a time, without access to the internal structure.

Language

Array Nothing methods exposed by Array.

Exceptions `throw new Error("message")` throws an error and halts the current thread. Uncaught errors propagate up the call stack and terminate the program if they reach the top. Exceptions are used instead of return codes to keep error handling separate from normal logic. Unchecked exceptions (`RuntimeException`) do not need to be declared in method signatures.

Exceptions are not subclassed.

Interfaces A way to create a type that lists methods that are guaranteed to exist. Classes can implement multiple interfaces.

Interfaces can extend other interfaces.

Record class A lightweight class for immutable data. Fields are declared in the constructor and automatically exposed as read-only accessors. Equivalent to a struct in other languages. Records can implement interfaces but cannot extend other classes.

```
record Point(int x, int y) {
    // x() and y() are auto-generated
    // toString() prints Point[x=..., y=...]
}
```

Parameter polymorphism Parameterized types let a class or interface take other types as arguments, enabling reusable code that works with any type. Generics are erased at compile time — the compiler uses type information to check correctness, but the runtime sees only raw types. Primitives and arrays cannot be used as type arguments. Use wrapper classes (`Integer`, `Boolean`) for primitive types.

```
class Box<T> {
    T value;
    Box(T v) { this.value = v; }
    T get() { return value; }
}

Box<String> s = new Box<>("hello");
Box<Integer> i = new Box<>(42);
```

Generic type bounds A bound lets you declare a type parameter must extend a specific type, giving the compiler enough information to call methods on it.

```
class NumBox<T extends Number> {
    T value;
    NumBox(T v) { this.value = v; }
    double doubleValue() { return value.doubleValue(); }
}

NumBox<Integer> n = new NumBox<>(5);
n.doubleValue();
```

Nested class A class defined inside another class. Static nested classes can access the outer class's static members. Non-static nested classes can access all members of the outer class including private ones.

```
class Outer {
    int x = 10;
    static int y = 20;

    class Inner {
        int getX() { return x; }
    }

    static class StaticInner {
        int getY() { return y; }
    }
}

Outer.Inner i = new Outer().new Inner();
Outer.StaticInner s = new Outer.StaticInner();
```

Anonymous class Defines a subclass or implementation inline, without a named class. Useful for one-off implementations. Can access fields of the enclosing class and local variables marked `final`.

```
interface Greeter {
    void greet();
}

Greeter g = new Greeter() {
    public void greet() {
        System.out.println("hello");
    }
};
g.greet();
```

Generic type bounds A bound can be a class or an interface. Multiple bounds are separated by `&`. The first bound can be a class, the rest must be interfaces.

```
class Box<T extends Number & Comparable<T>> {
    T value;
    Box(T v) { this.value = v; }
    int compareTo(T other) {
        return value.compareTo(other);
    }
}

Box<Integer> b = new Box<>(5);
b.compareTo(new Integer(3));
```

Wildcards Wildcards let you write code that works with any instance of a generic type, regardless of what type it holds. Use `extends` when you only read from the collection, `super` when you only write to it.

```
// reads only — can pass any Box<Number> or subclass
void printAll(Box<? extends Number> b) {
    System.out.println(b.get());
}

// writes only — can pass any Box<Integer> or superclass
void addOne(Box<? super Integer> b) {
    b.put(1);
}
```

Polymorphism Polymorphism lets you write code against a parent type and have the right method run automatically depending on the actual object at runtime. Subclasses override parent methods to provide their own behavior.

```
class Animal {
    void speak() {
        System.out.println("...");
    }
}

class Dog extends Animal {
    void speak() {
        System.out.println("woof");
    }
}

class Cat extends Animal {
    void speak() {
        System.out.println("meow");
    }
}

Animal[] zoo = new Animal[3];
zoo[0] = new Animal();
zoo[1] = new Dog();
zoo[2] = new Cat();

for (Animal a : zoo) {
    a.speak();
}
// ...
// woof
// meow
```

Inheritance Inheritance lets a subclass inherit fields and methods from a parent class, and override specific methods to change behavior. Use `extends` to create a subclass. Constructors are not inherited — the subclass must call `super(...)` to initialize the parent.

```
class Animal {
    int x = 1;
    void speak() {
        System.out.println("...");
    }
    Animal(int v) {
        this.x = v;
    }
}

class Dog extends Animal {
    Dog(int v) {
        super(v);
    }
    void speak() {
        System.out.println("woof");
    }
    void fetch() {
        System.out.println("fetch");
    }
}

Animal a = new Animal(5);
Dog d = new Dog(5);
d.speak();
d.fetch();
```

Abstract class Cannot be instantiated directly. Subclasses must implement all abstract methods. Can also have concrete methods and fields.

```
abstract class Shape {
    abstract double area();
    abstract double perimeter();
    void print() {
        System.out.println("shape");
    }
}

class Circle extends Shape {
    double r;
    Circle(double r) {
        this.r = r;
    }
    double area() {
        return 3.14 * r * r;
    }
    double perimeter() {
        return 2 * 3.14 * r;
    }
}

Shape s = new Shape();
Circle c = new Circle(5);
c.print();
```

Static Static fields belong to the class, not to any instance. Static methods can be called without an instance. Static initialization blocks run once when the class is first loaded.

```
class Counter {
    static int count = 0;
    static int getCount() {
        return count;
    }
    static {
        System.out.println("loaded");
    }
    Counter() {
        count++;
    }
}

Counter.getCount();
Counter c = new Counter();
Counter.getCount();
```

Final Prevents reassignment of a variable, field, or method parameter. Primitives are immutable, but reference types can still have their contents modified.

```
final int x = 5;
x = 6;

final int[] arr = {1, 2, 3};
arr[0] = 99;
arr = new int[3];
```

Equality `==` compares memory addresses for reference types and values for primitives. Use `.equals()` to compare contents. Override `equals` and `hashCode` together when defining custom equality.

```
int a = 5;
int b = 5;
a == b;

int[] x = {1, 2};
int[] y = {1, 2};
x == y;
x.equals(y);

String s = "hello";
String t = "hello";
s == t;
s.equals(t);
```

Casting Casting converts a value from one type to another. Upcasting is safe and automatic. Downcasting requires an explicit cast and may fail at runtime if the object is not actually the target type.

```
Animal a = new Dog();
Dog d = (Dog) a;
d.fetch();

int x = 5;
double y = (double) x;
int z = (int) y;
```

Exceptions Exceptions are used instead of return codes to keep error handling separate from normal logic. Unchecked exceptions (`RuntimeException`) do not need to be declared in method signatures. Exceptions are not subclassed.

```
try {
    int[] arr = {1, 2, 3};
    arr[5] = 99;
    int x = 10 / 0;
    throw new Error("something");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("out of bounds");
} catch (ArithmeticException e) {
    System.out.println("divide by zero");
} catch (Error e) {
    System.out.println("error");
} finally {
    System.out.println("always runs");
}
```

Interfaces A way to create a type that lists methods that are guaranteed to exist. Classes can implement multiple interfaces. Interfaces can extend other interfaces.

```
interface Greeter {
    void greet();
    void greet(int times);
    int getCount();
}

class EnglishGreeter implements Greeter {
    int count = 0;
    public void greet() {
        System.out.println("hello");
        count++;
    }
    public void greet(int times) {
        for (int i = 0; i < times; i++) {
            greet();
        }
    }
    public int getCount() {
        return count;
    }
}

Greeter g = new EnglishGreeter();
g.greet();
g.greet(3);
g.getCount();
```

Record class A lightweight class for immutable data. Fields are declared in the constructor and automatically exposed as read-only accessors. Equivalent to a struct in other languages. Records can implement interfaces but cannot extend other classes.

```
record Point(int x, int y) {
    // x() and y() are auto-generated
    // toString() prints Point[x=..., y=...]
}

Point p = new Point(3, 4);
System.out.println(p);
System.out.println(p.x());
System.out.println(p.y());
// Point[x=3, y=4]
// 3
// 4
```

Parameter polymorphism Parameterized types let a class or interface take other types as arguments, enabling reusable code that works with any type. Generics are erased at compile time — the compiler uses type information to check correctness, but the runtime sees only raw types. Primitives and arrays cannot be used as type arguments. Use wrapper classes (`Integer`, `Boolean`) for primitive types.

```
class Box<T> {
    T value;
    Box(T v) {
        this.value = v;
    }
    T get() {
        return value;
    }
}

Box<String> s = new Box<>("hello");
Box<Integer> i = new Box<>(42);
```

Generic type bounds A bound lets you declare a type parameter must extend a specific type, giving the compiler enough information to call methods on it.

```
class NumBox<T extends Number> {
    T value;
    NumBox(T v) {
        this.value = v;
    }
    double doubleValue() {
        return value.doubleValue();
    }
}

NumBox<Integer> n = new NumBox<>(5);
n.doubleValue();
```

Nested class A class defined inside another class. Static nested classes can access the outer class's static members. Non-static nested classes can access all members of the outer class including private ones.

```
class Outer {
    int x = 10;
    static int y = 20;

    class Inner {
        int getX() {
            return x;
        }
    }

    static class StaticInner {
        int getY() {
            return y;
        }
    }
}

Outer.Inner i = new Outer().new Inner();
Outer.StaticInner s = new Outer.StaticInner();
```

Anonymous class Defines a subclass or implementation inline, without a named class. Useful for one-off implementations. Can access fields of the enclosing class and local variables marked `final`.

```
interface Greeter {
    void greet();
}

Greeter g = new Greeter() {
    public void greet() {
        System.out.println("hello");
    }
};
g.greet();
```

Generic type bounds A bound can be a class or an interface. Multiple bounds are separated by `&`. The first bound can be a class, the rest must be interfaces.

```
class Box<T extends Number & Comparable<T>> {
    T value;
    Box(T v) {
        this.value = v;
    
```