Object-Oriented Programming

Lecture 4: Inheritance

Sebastian Junges

"Inheritance is the most prominent and most overused part of OO"

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

Goal Today

- You can work with inheritance:
  You can define subclasses, call the constructor of a superclass, …
- You can discuss use cases and differences between
  (abstract) classes and interfaces, using is-a and has-a relationships
- You recognize which method is called, using the difference between
  overloading, overriding, and hiding
- You have an elementary understanding of exception handling

Example: Bike

Idea: create an interface and two classes

- State
  - color,
  - size,
  - wheels,
  - frameMaterial
- Behavior:
  - getColor(),
  - getSize(),
  - getWheels(),
  - getFrameMaterial()
- Now we aim to extend the class to bikes that can shift

Example: The **`Bike`** interface

Idea: create an interface and two classes

```
public interface Bike {
    // Instead of strings, use dedicated types...

    public String getFrameMaterial();
    public int getSize();
    public String getColor() ;
    public int getSpeed();
    + Setters

}
```

Problem: Leads to plenty of code duplication

Extending classes

- A bike with gears **is a** bike with:
  - additional fields for the state
  - additional behavior
  - an updated behavior

- If a function expects a bike, passing a bike with gears is also fine

The `Bike` (base) class

```java
public class Bike {
    // Instead of strings, use dedicated types...
    private String frameMaterial; private int size; private String color; private int speed;

    public Bike(String frameMaterial, int size, String color, int speed) {
        this.frameMaterial = frameMaterial;
```

+ set other fields

```java
    }

    public String getFrameMaterial() { return frameMaterial; }
```

+ Getters/Setters

```java
    public String toString() {
        return "Bike{" + "frameMaterial='" + frameMaterial + '\'' +
                    ", size=" + size + ", color='" + color + '\’’ + ", speed=" + speed + '}';
    }
}
```

The BikeWithGears (derived) class

```java
public class BikeWithGears extends Bike {
    private int numberGears;

    public BikeWithGears(String frameMaterial, int size,
                         String color, int speed, int numberGears) {
        ??    (comes later)
    }

    public int getNumberGears() { return numberGears; }

    @Override
    public String toString() {
        ??    (comes later)
    }
}
```

Using Bike and BikeWithGears

```java
public static void main(String[] args) {
    Bike b = new Bike("Steel", 59, "Red", 16);
    System.out.println(b.toString());

    BikeWithGears bg = new BikeWithGears("Carbon", 57, "Blue", 18, 7);
    System.out.println(bg.toString());

    bg.setColor("Green");
    System.out.println(bg.toString());

    Bike b2 = bg;
    System.out.println(b2.toString());
}
```

```
Compile & Run Lecture4:
Bike{framematerial=‘Steel’, …}
BikeWithGears{framematerial=‘Carbon’,…, nrGears=7}
BikeWithGears{framematerial=‘Carbon’,…, nrGears=7}
BikeWithGears{framematerial=‘Carbon’,…, nrGears=7}
Success
```

Inheritance

- A base class can be extended by a derived class
  - Any class can be a base class
- The derived class has at least the same state and behavior…
- .. but can have additional fields
- .. and can have additional behavior
- .. and change the behavior of the base class
- Derived classes define subtypes

Content Today

- Inheritance
  - Derived classes
  - Constructors in derived classes, including `super(…)`
  - Delegating work (e.g., when converting to a string), including `super.`
  - The protected access modifier
  - Final classes and methods
- Other stuff

Defining derived classes

```java
public class Base {
    private int x;
    public Base() { this.x = 3; }
}

public class Derived extends Base {
    private int y;
    public Derived (int y) { this.y = y; }
}
```

Derived classes without default constructor

```
public class Base {
      private int x;
      public Base(int x) { this.x = x; }
}

public class Base extends Derived {
      private int y;
      public Derived (int x, int y) {
            // what should come here?
            super(x); this.y = y;
      }
}
```

The `super(...)` constructor invocation

- `super(…)` calls the constructor of the (unique) superclass
- If `super(…)` is not called explicitly, the compiler adds a call to `super()`

The `Bike` (base) class

```
public class Bike {
    // Instead of strings, use dedicated types...
    private String frameMaterial; private int size; private String color; private int speed;

    public Bike(String frameMaterial, int size, String color, int speed) {
        this.frameMaterial = frameMaterial;
```

+ set other fields

```
    }

    public String getFrameMaterial() { return frameMaterial; }
```

+ Getters/Setters

toString method

```
}
```

The `BikeWithGears` (derived) class

```java
public class BikeWithGears extends Bike {
    private int numberGears;

    public BikeWithGears(String frameMaterial, int size,
                         String color, int speed, int numberGears) {
        super(frameMaterial, size, color, speed);
        this.numberGears = numberGears;
    }

    public int getNumberGears() { return numberGears; }

    @Override
    public String toString() {

        ??    (comes later)

    }
}
```

Content Today

- Inheritance
  - Derived classes
  - Constructors in derived classes, including `super(…)`
  - Delegating work (e.g., when converting to a string), including `super.`
  - The protected access modifier
  - Final classes and methods
- Other stuff

`BikeWithGears.toString()` Attempt 1

- Idea: Call `Bike.toString()` inside `BikeWithGears.toString()`
- How? We need to access `toString` from a base class.
- Solution. The `super` keyword

The super keyword

- Analogous to `this`, but for the superclass
- Helpful for calling overriden methods

```
public class BikeWithGears extends Bike {
  @Override
  public String toString() {
    return "BikeWithGears{" +
    super.toString() + ", numberGears=" + numberGears +  "}";
  }
}
```

`BikeWithGears.toString()` Attempt 2

- Downside in Attempt 1: we actually want to change the behavior of the base class but still reuse parts of the code.

Delegating Work to Derived Classes (1)

- Idea: `toString` for bikes prints the type of bike and a list of fields.

```
public class Bike {
    public String getBikeTypeAsString()  { return "Bike"; }

    public String getBikeAttributesAsString() {
        return "frameMaterial='" + frameMaterial + '\'' +  ", size=" + size +
               ", color='" + color + '\'' + ", speed=" + speed;
    }

    @Override
    public String toString() {
        return getBikeTypeAsString() + "{" + getBikeAttributesAsString() + '}';
    }
}
```

Delegating Work to Derived Classes (2)

- Idea: `toString` for bikes prints the type of bike and a list of fields.

```
public class BikeWithGears {
    @Override
    public String getBikeTypeAsString() {
        return "BikeWithGears";
    }

    @Override
    public String getBikeAttributesAsString() {
        return super.getBikeAttributesAsString() + " , numberGears=" + numberGears
    }
}
```

Content Today

- Inheritance
  - Derived classes
  - Constructors in derived classes, including `super(…)`
  - Delegating work (e.g., when converting to a string), including `super.`
  - The protected access modifier
  - Final classes and methods
- Other stuff

The protected access modifier (revisited)

- We cannot override private methods*
- Recall that between private and public, the protected keyword exists: protected allows access from:
  - subclasses
  - classes in the same package **

- Be careful with protected. Use private whenever possible.

* This is different in other OO languages like C++ and leads to some differences in design

** This is different in other OO languages like C# and C++ and leads to some differences in design.

The `final` keyword

- For fields: constants
- For methods: prevent overriding
- For classes: prevent deriving

- Prevents errors if you did not design the class for inheritance
- Important to make a class inmutable (details are not discussed).

Example: Final classes

```java
public final class FinalBase {

}

public class DeriveFinal extends FinalBase {

}
```

```
Compile Lecture4:

java: cannot inherit from final class FinalBase
Error
```

Example: Final methods

```java
public class Base {
    public void aMethod() { System.out.println("Base::aMethod"); }

    public final void bMethod() { System.out.println("Base::bMethod"); }
}

public class Derived extends Base {
    @Override
     public void aMethod() { System.out.println("Derived::aMethod"); }

    @Override
    public void bMethod() { Sy
}
```

```
Compile Lecture4:

java: Derived cannot override bMethod() in
lecture4.Base overridden method is final
Error
```

Content Today

- Inheritance
  - Derived classes
  - Constructors in derived classes, including `super(…)`
  - Delegating work (e.g., when converting to a string), including `super.`
  - The protected access modifier
  - Final classes and methods
- Other stuff

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

Recap: Late binding of `this`

```
public static void main(String[] args) {
    EnergyConsumer tv = new TV(32);
    System.out.println(tv.getRequiredVoltage());
    EnergyConsumer heater = new Heater();
    System.out.println(heater.getRequiredVoltage());
}
```

```
Compile & Run Lecture3:

230
380

Success
```

Recap: Early binding of parameters

```java
public static void main(String[] a) {
    Heater h1 = new Heater();
    whatAmI(h1);
    EnergyConsumer h2 = new Heater();
    whatAmI(h2);

}
```

```java
public static void whatAmI(EnergyConsumer ec) {
    System.out.println("an Energyconsumer");
}
```

```java
public static void whatAmI(Heater h) {
    System.out.println("a Heater");
}
```

```
Compile & Run Lecture3:

a Heater
an EnergyConsumer

Success
```

Overloading vs Overriding vs Hiding (1)

- Method signature: methodName (<ParameterTypes>)
  - not the return type!
- Overriding: Define a method with the same signature in a subclass that is called via late binding
  - Cannot work for static or private methods
  - Matching: Return type can be a subtype
- Overloading: Define a method with same name but different parameter types
- Hiding: Define a variable (or method) with the same name (or signature)

Overloading vs Overriding vs Hiding (2)

This is overriding!

```
public class Base {
    public void bMethod(Bike b) { System.out.println("Base::b"); }
}

public class Derived extends Base {

    public void bMethod(Bike b) { System.out.println("Derived::b"); }
}

public static void main(String[] args) {
    Bike b = new Bike(…);
    BikeWithGears bg = new BikeWithGears(…);
    Derived test = new Derived();
    test.bMethod(b);
    test.bMethod(bg);
}
```

```
Compile Lecture4:
Derived::b
Derived::b
Success
```

Overloading vs Overriding vs Hiding (3)

This is overloading!
and early binding

```
public class Base {
    public void bMethod(BikeWithGears b) { System.out.println("Base::b"); }
}

public class Derived extends Base {

    public void bMethod(Bike b) { System.out.println("Derived::b"); }
}

public static void main(String[] args) {
    Bike b = new Bike(…);
    BikeWithGears bg = new BikeWithGears(…);
    Derived test = new Derived();
    test.bMethod(b);
    test.bMethod(bg);
}
```

```
Compile Lecture4:
Derived::b
Base::b
Success
```

Overloading vs Overriding vs Hiding (4)

```java
public class Base {
    public void bMethod(BikeWithGears b) { System.out.println("Base::b"); }
}

public class Derived extends Base {
    @Override
    public void bMethod(Bike b) { System.out.println("Derived::b"); }
}

public static void main(String[] args) {
    Bike b = new Bike(…);
    BikeWithGears bg = new BikeWi
    Derived test = new Derived()
    test.bMethod(b);
    test.bMethod(bg);
}
```

```
Compile Lecture4:
java: method does not override or implement a method
from a supertype

Error
```

Overloading vs Overriding vs Hiding (5)

```java
public class Base {
    public void cMethod( ) { System.out.println("Base::cMethod");}
    public void dMethod() { cMethod(); }
    public void eMethod() { cMethod(); }
}
public class Derived extends Base {
    public void cMethod() { System.out.println("Derived::cMethod"); }
    public void dMethod() { cMethod(); }
}


public static void main(String[] args) {
    Base base = new Base(); Derived der = new Derived();
    base.dMethod();
    base.eMethod();
    der.dMethod();
    der.eMethod();
}
```

This is overriding!
and late binding

```
Compile Lecture4:
Base::cMethod
Base::cMethod
Derived::cMethod
Derived::cMethod
Success
```

Example Overloading vs Overriding (6)

```
public class Base {
    private void cMethod( ) { System.out.println("Base::cMethod");}
    public void dMethod() { cMethod(); }
    public void eMethod() { cMethod(); }
}

public class Derived extends Base {
    private void cMethod() { System.out.println("Derived::cMethod"); }
    public void dMethod() { cMethod(); }
}


public static void main(String[] args) {
    Base base = new Base(); Derived der = new Derived();
    base.dMethod();
    base.eMethod();
    der.dMethod();
    der.eMethod();
}
```

This is not overriding (private)!
It is hiding

```
Compile Lecture4:
Base::cMethod
Base::cMethod
Derived::cMethod
Base::cMethod
Success
```

Avoid accidental hiding/overloading

- Use @Override
- Do not hide (except local variables in constructors and setters)

"Compiler errors are better than runtime crashes"
Whenever possible, build code that does not compile if misused

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

Abstract classes

- Classes that should not create objects (like interfaces)
  - have fields (unlike interfaces)
  - have constructors (!)
- “Incomplete classes”
  - describe a contract like interfaces
  - implemented methods can use this contract
    (recall the toString example for bikes)
- Classes that are not abstract are concrete

Example: Abstract Animal Class

```
public abstract class Animal {
    private double typicalLifeSpan;
    private double typicalWeight;

    public Animal(double typicalLifeSpan, double typicalWeight) {
        this.typicalLifeSpan = typicalLifeSpan;
        this.typicalWeight = typicalWeight;
    }

    public abstract String makeSound();

    public double getTypicalLifeSpan() {
        return typicalLifeSpan;
    }

    public double getTypicalWeight() {
        return typicalWeight;
    }
}
```

Example: Abstract Animal Class (2)

```java
public class Duck extends Animal {
    // No constructors.

    @Override
    public String makeSound() {
        return "Quack";
    }
}
```

```
Compile Lecture4:
in Duck.java:
java: constructor Animal in class lecture4.Animal cannot be
applied to given types;
  required: double,double
  found:    no arguments
  reason: actual and formal argument lists differ in length
Error
```

Example: Abstract Animal Class (3)

```java
public class Duck extends Animal {

    Duck(double typicalLifeSpan, double typicalWeight) {
        super(typicalLifeSpan, typicalWeight);
    }

    @Override
    public String makeSound() {
        return "Quack";
    }
}
```

Abstract classes vs Interfaces

- Abstract classes have fields, constructors and can have private methods
- A class can implement multiple interfaces
- One cannot create objects of the abstract class nor of the interface
- “Prefer interfaces over abstract classes”

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

Inheritance & Type Hierarchies

- Introduce a hierarchy of types
- Both on classes and interfaces
- From most abstract to more concrete

Example: Hierarchy (1)

```java
public abstract class Animal {
    private double typicalLifeSpan;
    private double typicalWeight;

    public Animal(double typicalLifeSpan,double typicalWeight) {
        this.typicalLifeSpan = typicalLifeSpan;
        this.typicalWeight = typicalWeight;
    }

    public abstract String getSound();
    public double getTypicalWeight() { return typicalWeight; }
}

public interface CanFly {
    default void fly() { System.out.println("Flapflap"); }
}

public abstract class Bird extends Animal implements CanFly {
    public Bird(double typicalLifeSpan, double typicalWeight) {
        super(typicalLifeSpan, typicalWeight);
    }
}
```

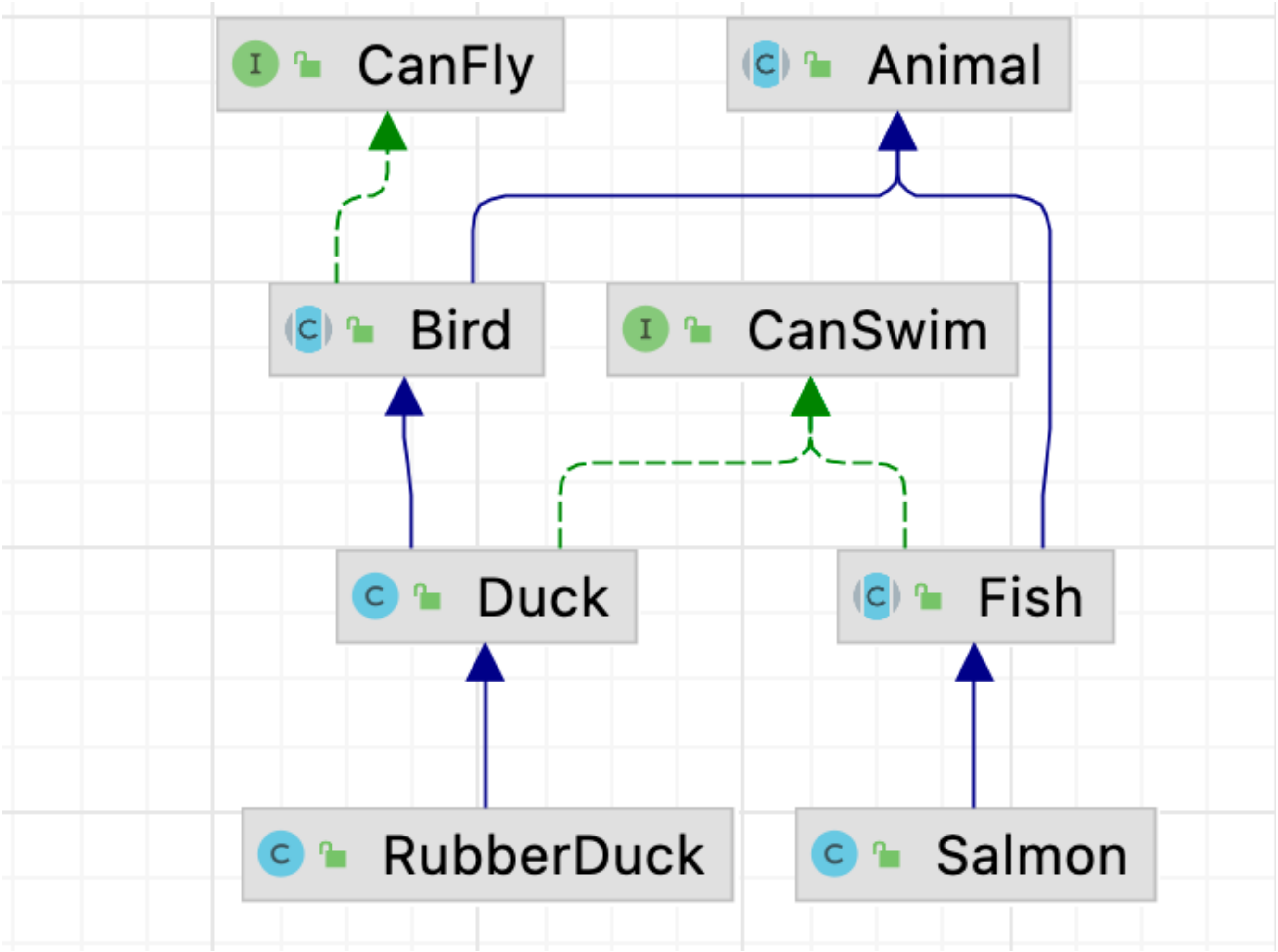
Example: Hierarchy (2)

```java
public class Duck extends Bird implements CanSwim {
    Duck(double typicalLifeSpan, double typicalWeight) {
        super(typicalLifeSpan, typicalWeight);
    }

    @Override
    public String makeSound() { return "Quack"; }
}

public static void main(String[] args) {
    Duck mallard = new Duck(6, 2.2);
    mallard.fly();
    mallard.swim();
    System.out.println(mallard.makeSound());
}
```

Example: Hierarchy (3)



`Object` class

- An implicit base class for every class (direct or indirect)
- Four important methods
  - `String toString()` … No particular requirements
  - `boolean equals(Object other)` … Next slide
  - `int hashCode()` ... Important when using Maps, later
  - `Object clone()` … see Q&A

Implementing `bool Equals(Object other)`

- Equality should satisfy the following for non-null `x,y,z`
  - *reflexive* `x.equals(x)` returns true
  - *symmetric* `x.equals(y) == y.equals(x)`
  - *transitive* `(x.equals(y)` and `y.equals(z))` implies `x.equals(z)`
  - *consistent* `x.equals(y)` returns the same value over multiple invocations
  - `x.equals(null)` returns false

- Check whether types are consistent
- Make sure to stick to the signature
- Tricky if a superclass is not abstract
- Only override equals if necessary

Example: `boolean Equals(Object o)`

```
@Override
public boolean equals(Object other) {
    // Often done, good for e.g. search or sorting
    if (this == other) { return true; };
    if !(other instanceof Bike) {
        return false;
    }
    Bike oBike = (Bike) other;
    return frameMaterial.equals(oBike.frameMaterial) && size == other.size && …;
}
```

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

How to model complex systems?

- Disclaimer: No general rule
  - Depends on the context.
  - Depend on the type of relationship between objects

Strong Is-a relationships

- A bike with gears **IS A** bike
- A door with a lock **IS A** door
- A student **IS A** person
- Ok to use inheritance

Has-a relationships

- A person has an address
- A door has a lock
- A bike has a color

- Use composition or interfaces (see next slide)

Weak is-a (Is-a-kind-of) relationships

- TVs are a kind of energy consumer
- Credit cards are a kind of payment method

- Typically has-a property, but not has-a state, or when a thing is multiple things…
- Generic rule: Use Interfaces

Multiple Inheritance and Mixins

- Consider a class `Singer` and a class `Songwriter`.
  What about singer-songwriters?

- Java classes can extend one base class only
  (and implement many interfaces)
- “There is no multiple inheritance in Java”
- Mixin: Primary type + secondary types from interfaces:
  - e.g., a Singer-Songwriter IS A singer that can songwrite (mostly)

Skeleton Implementations

- Define an interface CanSing and an abstract class AbstractSinger that implements CanSing
- Define an interface CanSongwrite and an abstract class AbstractSongwriter that implements CanSongwrite
- The class Singer can now extend AbstractSinger with minimal effort
- The class Songwriter can now extend AbstractSongwriter…
- The class SingerSongwriter extends AbstractSinger and implements CanSongwrite…
- The Java API has plenty of Skeleton Implementations, e.g. AbstractList

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

OO Fundamentals (Overall considerations)

- Any code/functionality exposed to a user will be used
  - Only expose the code that you need to expose
  - It is super hard to figure out whether a bug is your fault or the user’s fault
- Code bases need to be maintained
  - Often after years
  - Often by other people

OO Fundamentals (Design Goals)

- Objects have a state and a behavior
  - Classes describe objects (can be abstract, conceptual or concrete)
- Encapsulation
  - Change the state only via behavior
  - Hide the implementation types
  - Identify what varies and separate that from what stays the same (see composition)
- Separation of Concerns
  - Classes should do one thing and one thing only
  - IO is always a thing. Thus, if a class does IO, it shouldn’t do more.
- Loose Coupling
  - Program to an interface, not an implementation
  - Allow to replace and extend your code

OO Fundamentals (Technical ingredients)

- (Object) Polymorphism & Class Hierarchies
  - Objects have one dynamic type, this type can be a subtype of other types
- Late Binding (also: Method Polymorphism)
  - Method calls depend on the dynamic type of this
  - In contrast, parameter types are staticly determined (early binding)
- Composition & Inheritance
  - Do not solve everything with inheritance
  - Inheritance for is-a, composition for has-a. Is-a-kind-of typically via interfaces

Content Today

- Inheritance
- Method selection: Overriding vs Overloading vs Hiding
- Abstract classes (vs Interfaces)
- Class hierarchies and the Object class
- Interfaces or inheritance: Is-a vs has-a relationships
- OO Fundamentals Recap
- *Exception* handling

Exceptions

- When a program runs into a runtime error, the program terminates abruptly.
- How can you handle the runtime error so that the program can continue to run or terminate gracefully?
- Answer: by using (Java) Exceptions and Exception handling.
- If exceptions are not handled (explicitly), the program will terminate abruptly.

Exceptions (Examples)

```
Object obj = null; obj.toString();
```

```
Exception … java.lang.NullPointerException
```

```
int i = 4711/0;
```

```
Exception … java.lang.ArithmeticException: / by zero
```

```
int[] a = { 1, 2, 3 };
System.out.println( a[3] );
```

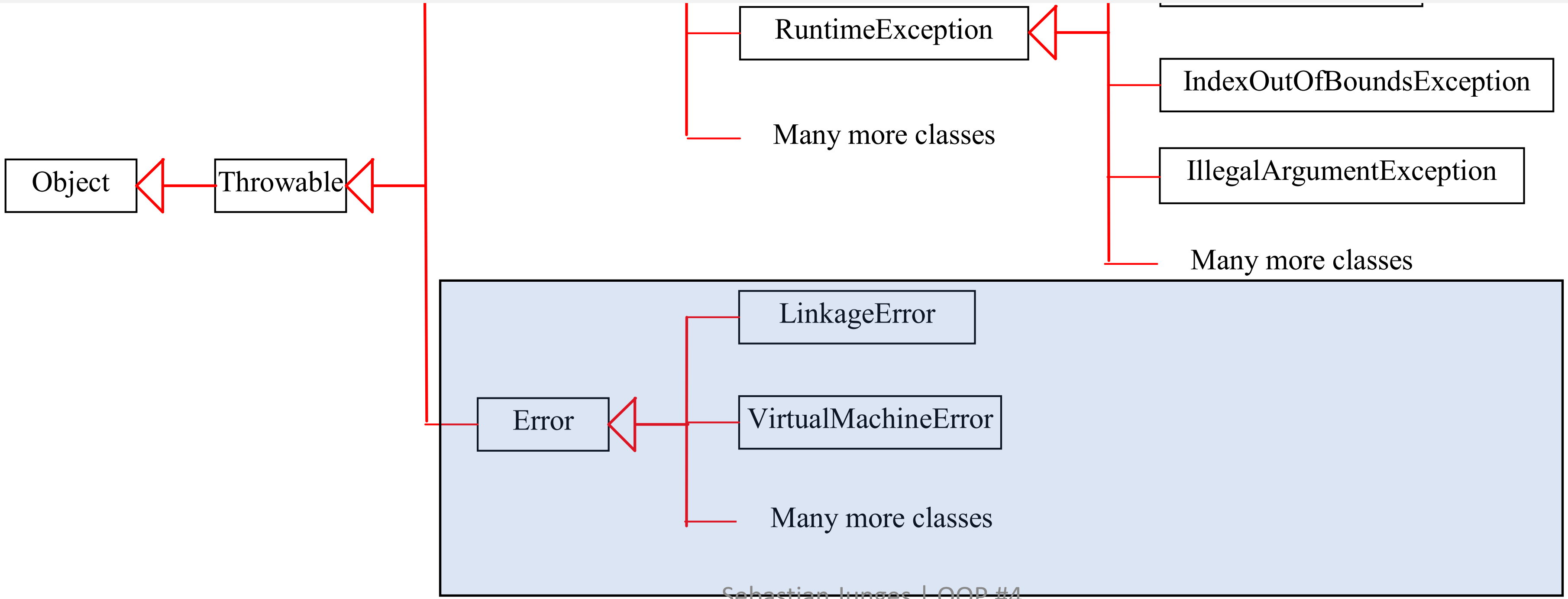
```
Exception … java.lang.ArrayIndexOutOfBoundsException:
Index 3 out of bounds for length 3
```

What is an exception?

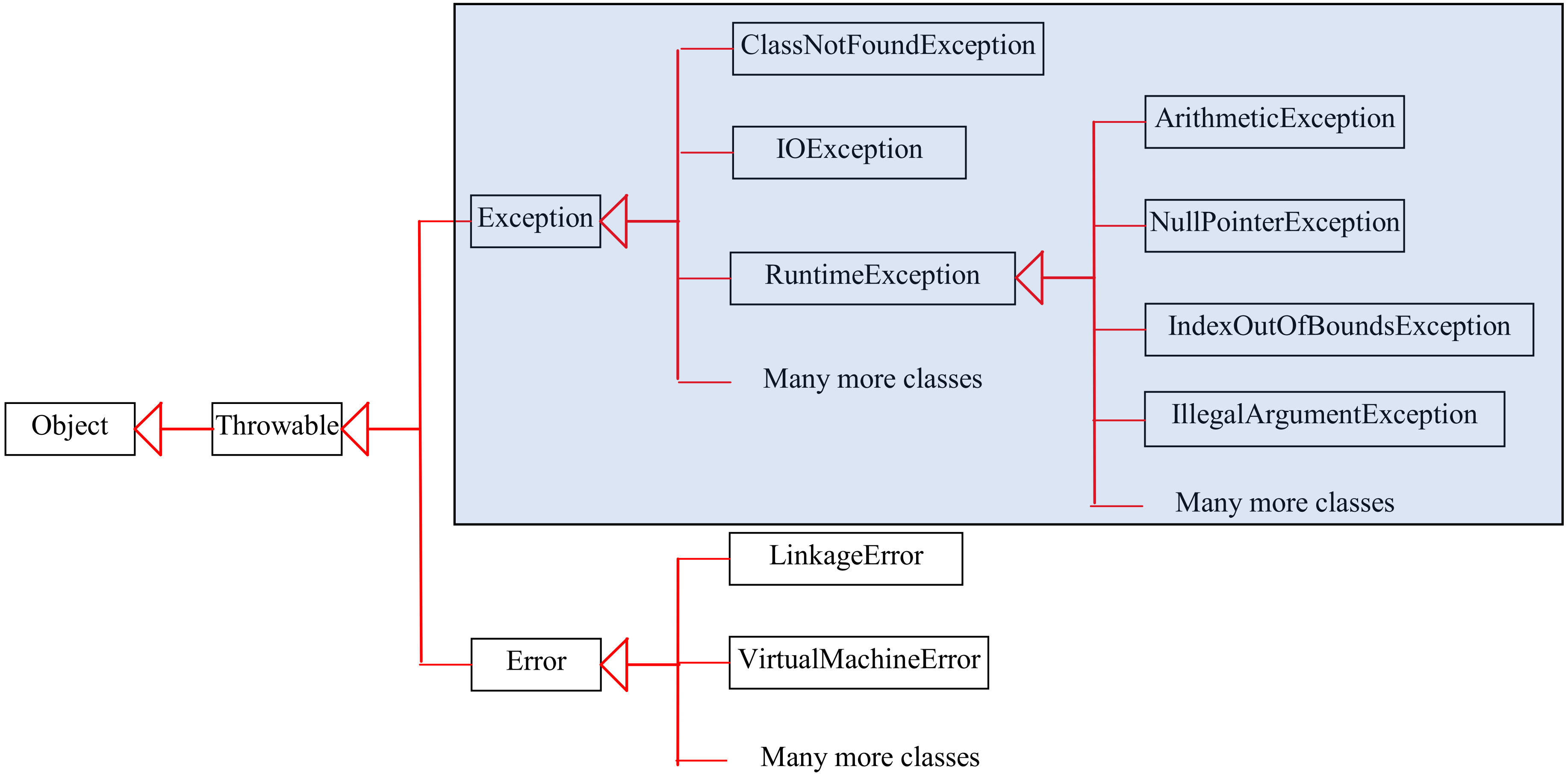
- In Java, runtime errors are thrown as exceptions
- An exception is an object that represents the kind of error.
- Java provides standard exception classes, such as
  - `Exception`
  - `RuntimeException`
- You can use these to generate exceptions yourself or to introduce new classes for your own error handling

Errors

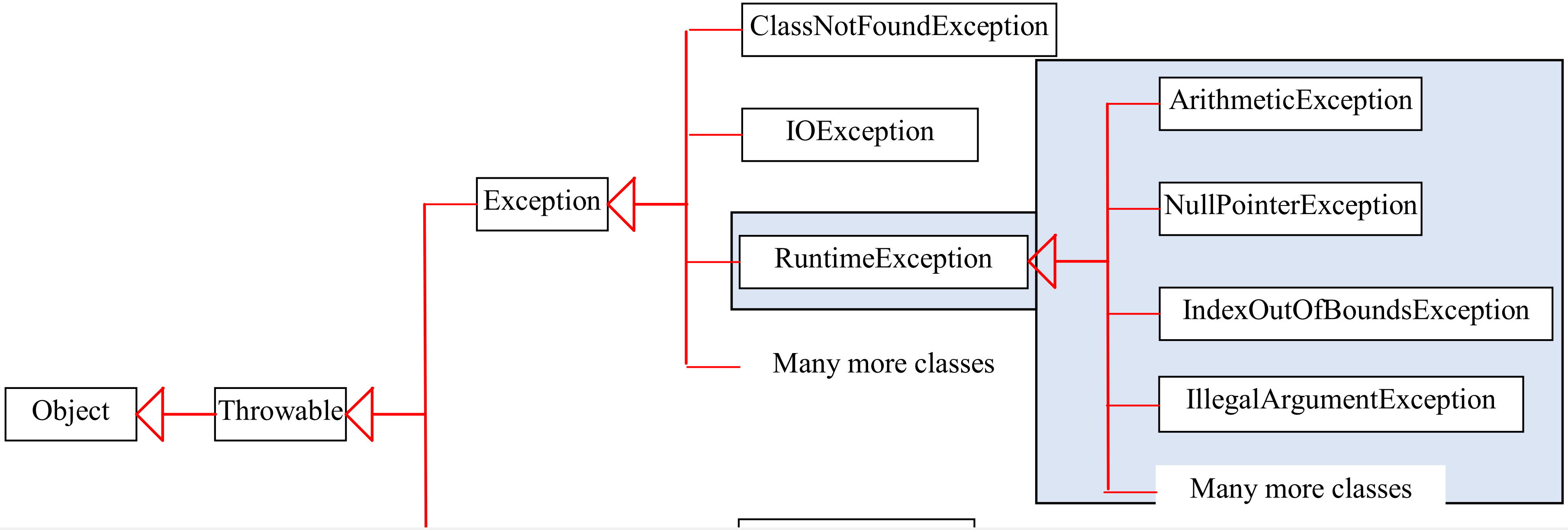
- `Errors` are due to issues in the JVM including out of memory issues



Exception Class Hierarchy



Runtime Exceptions



- `RuntimeExceptions` are caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Throwing Exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as throwing an exception.

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```

Declaring Exceptions

Methods must state the types of checked exceptions they may throw.
This is known as declaring exceptions.

```
public void myMethod() throws IOException { }
public void myMethod() throws IOException, OtherException { }
```

Throwing Exceptions Example

```
public void setRadius( double newRadius ) throws illegalArgumentException {
    if (newRadius >= 0) {
      radius =  newRadius;
    } else {
      throw new IllegalArgumentException( "Radius cannot be negative" );
    }
}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
} catch ( Exception1 exVar1 ) {
  handler for exception1;
} catch ( Exception2 exVar2 ) {
  handler for exception2;
}
...
} catch ( ExceptionN exVar3 ) {
  handler for exceptionN;
}
```

Checked and Unchecked Exceptions

`RuntimeException` and its subclasses are unchecked exceptions.

- for programming errors

All other exceptions are checked exceptions.

- for user or environment errors

The compiler enforces that one *deals* with checked exceptions (see next)

(Java) `Errors` are unchecked

Dealing with exceptions

If a method (see p2) declares a checked exception, you must invoke it in

- a try-catch block (see a), or
- declare to throw the exception in the calling method (see b).

```
void p2() throws IOException {
   if ( a file does not exist ) {
      throw new IOException( "File not found" );
   } }
```

(a)

```
void p1() { try {
              p2();
            } catch ( IOException ex ) {
              process exception
            } }
```

(b)

```
void p1() throws IOException {
   p2();
}
```

Exceptions: Design considerations

- Write understandable error messages
- Define errors on the correct abstraction level
  - E.g, internally an error may be out of bounds, but to the external user it should be an invalid input
  - Catch and rethrow
- Reuse standard exceptions like IllegalArgumentException, IllegalStateExceptions, NullPointerExceptions,…
- Use exceptions for exceptions only
- Do handle exceptions (no empty catch blocks)

OOP continues next week

Thanks!

I will see you next quarter for Lectures 12-14: Concurrency and OOP