Inheritance

Lecture 4 (16 february 2021)

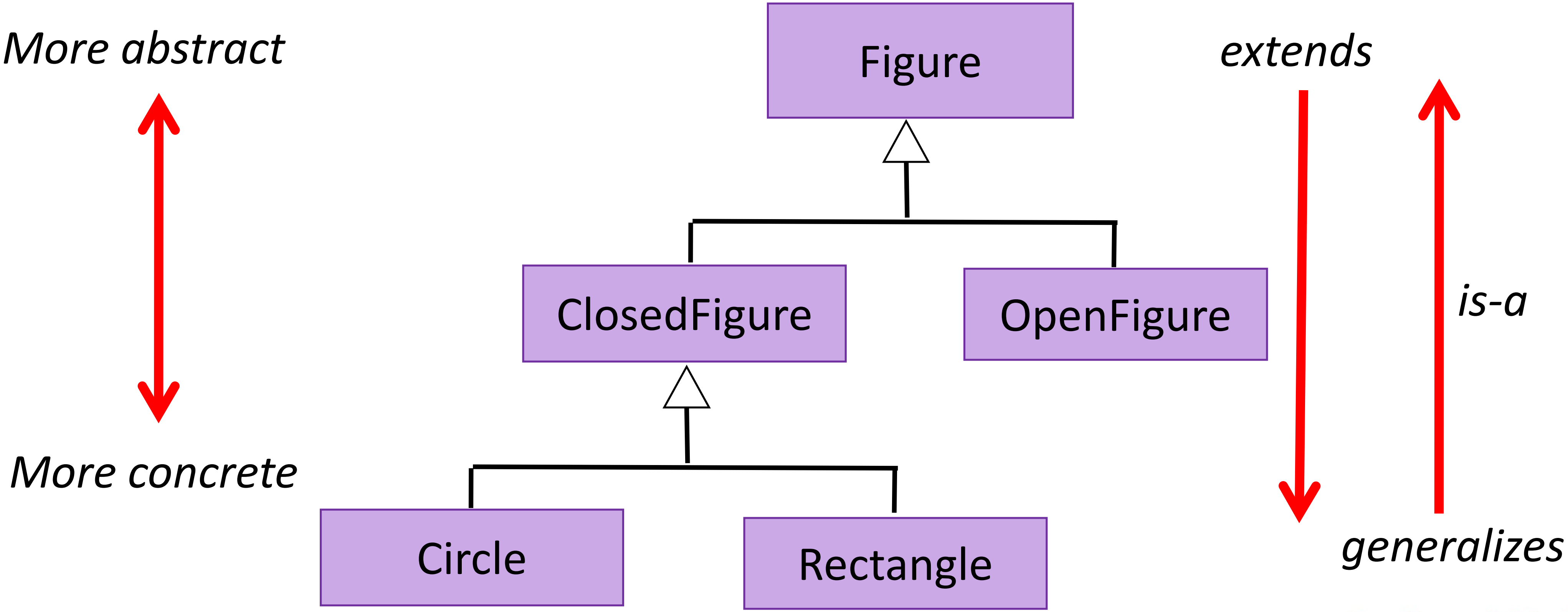
Radboud University

Abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!

Abstraction and classes

- Abstraction: less detail, less complex, more general
- Can define abstraction relation between classes.
- Can specify that one class generalizes or abstracts another class.

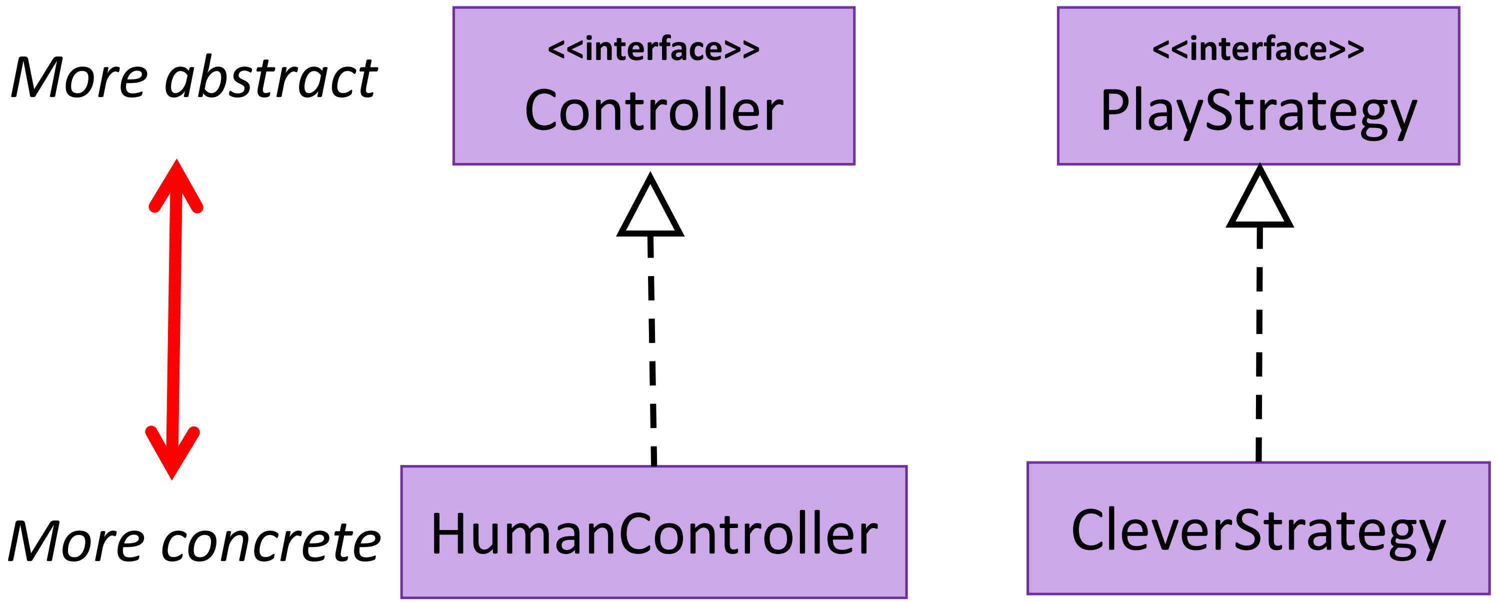


Abstraction and interfaces

- An abstraction can be realized with an interface.

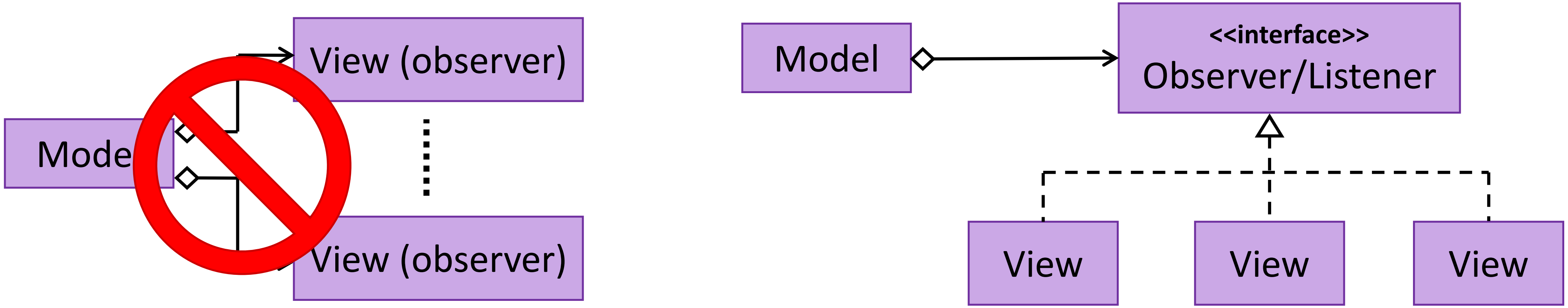
OOP rule 1: program to an interface

  - clients should refer to a more abstract level of a class, instead of refering to a concrete implementation.

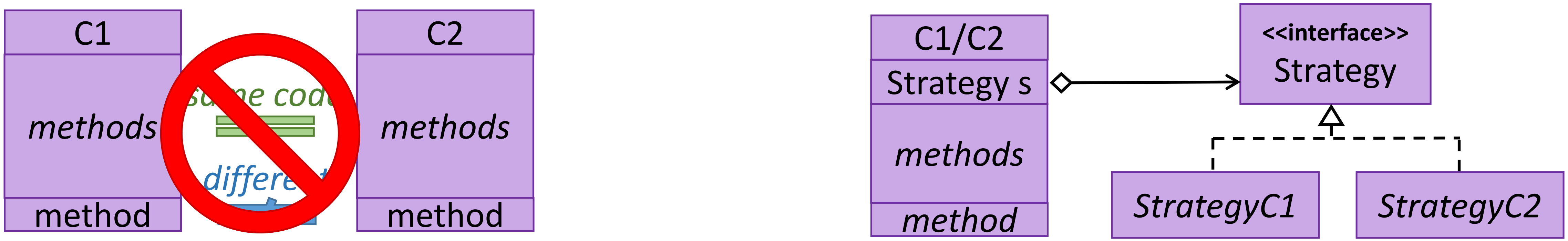


Interfaces and Design Patterns

- Interfaces are the core of design patterns (Lecture 10)

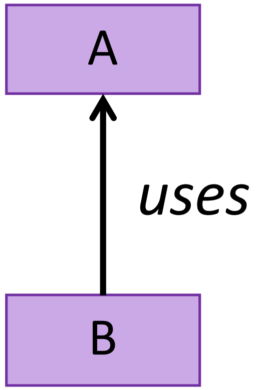


Observer Pattern



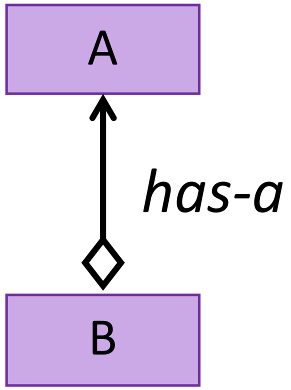
Strategy Pattern

Relations in (UML) class diagrams



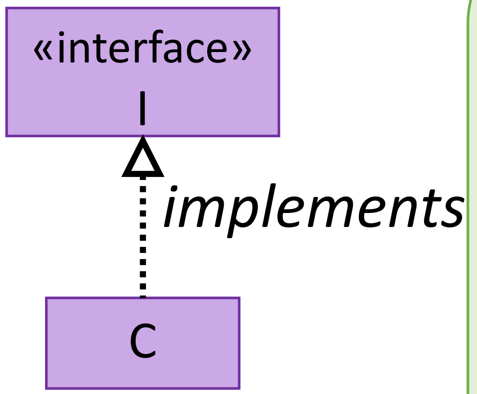
association

```
class B {
  public void mB( A a ) {
    a.mA();
  }
}
```



composition

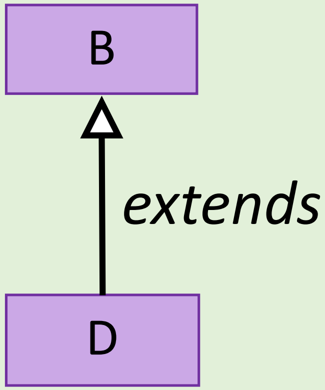
```
class B {
  private A attrOfB;
}
```



realization

```
interface I {}
class C implements I {}
```

today



inheritance

```
class B {}
class D extends B {}
```

Inheritance

- from abstract to more concrete -

- A new class is created from an existing class
  - Absorb existing class' data and behaviors (except the private methods)
  - Enhance with new capabilities
- The new class is called a derived class or subclass. The original class is called the base class or superclass.
- Subclass extends superclass
  - Subclass
    - More specialized group of objects
    - Behaviors inherited from superclass
    - Additional behaviors

Inheritance (2)

- A derived class (subclass) has
  - all attributes and (non private) methods of the base class (superclass)
  - additional methods and/or attributes
- When designing certain classes, there is often a natural hierarchy for grouping them.
- Object of one class "is an" object of another class
  - Example: Rectangle *is a* Quadrilateral (vierhoek).
  - Class Rectangle inherits from class Quadrilateral
  - Quadrilateral: superclass
  - Rectangle: subclass

Inheritance (3)

- Superclass typically represents *larger set of objects* than subclasses. Example:
  - *superclass: Vehicle*
    - Cars, trucks, boats, bicycles, …
  - *subclass: Car*
    - Smaller, more-specific subset of vehicles

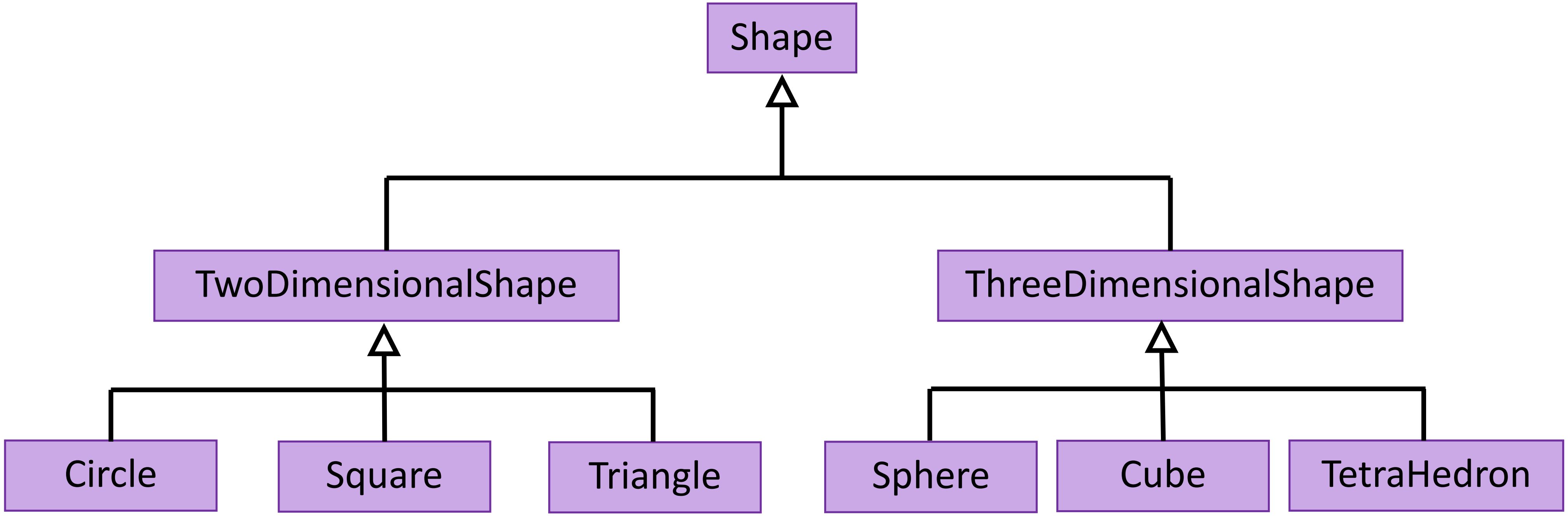
Class hierarchy

- Direct superclass
  - *one level up hierarchy*
- Indirect superclass
  - *Inherited two or more levels up hierarchy*
- Single inheritance
  - *Inherits from one superclass*
- Multiple inheritance
  - *Inherits from multiple superclasses*
    - Java does not support multiple inheritance

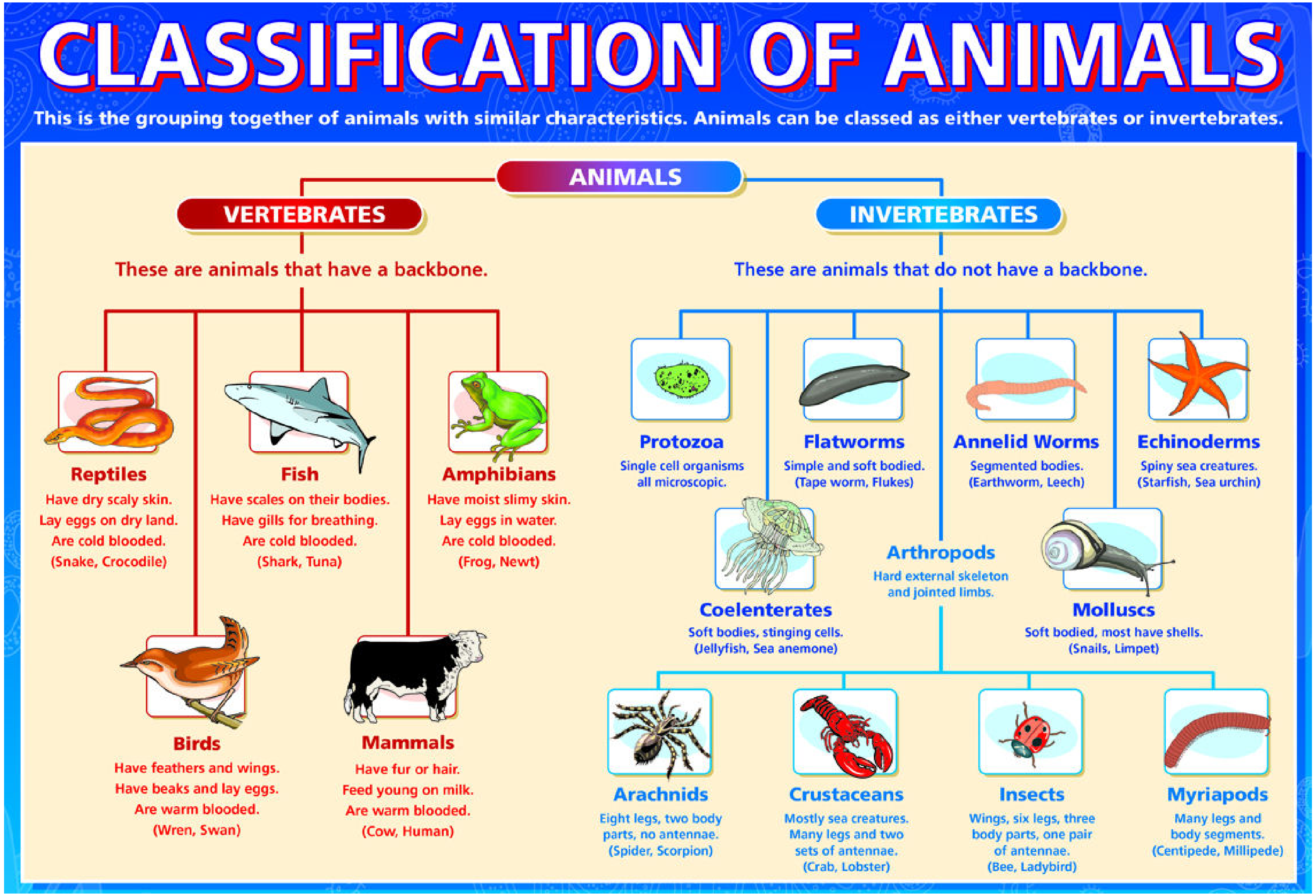
Inheritance examples.

Superclass	Subclass
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Inheritance hierarchy for Shapes.



Hierarchy for animals



CLASSIFICATION OF ANIMALS

This is the grouping together of animals with similar characteristics. Animals can be classed as either vertebrates or invertebrates.

ANIMALS

VERTEBRATES

These are animals that have a backbone.

Reptiles
Have dry scaly skin.
Lay eggs on dry land.
Are cold blooded.
(Snake, Crocodile)

Fish
Have scales on their bodies.
Have gills for breathing.
Are cold blooded.
(Shark, Tuna)

Amphibians
Have moist slimy skin.
Lay eggs in water.
Are cold blooded.
(Frog, Newt)

Birds
Have feathers and wings.
Have beaks and lay eggs.
Are warm blooded.
(Wren, Swan)

Mammals
Have fur or hair.
Feed young on milk.
Are warm blooded.
(Cow, Human)

INVERTEBRATES

These are animals that do not have a backbone.

Protozoa
Single cell organisms
all microscopic.

Flatworms
Simple and soft bodied.
(Tape worm, Flukes)

Annelid Worms
Segmented bodies.
(Earthworm, Leech)

Echinoderms
Spiny sea creatures.
(Starfish, Sea urchin)

Coelenterates
Soft bodies, stinging cells.
(Jellyfish, Sea anemone)

Arthropods
Hard external skeleton
and jointed limbs.

Molluscs
Soft bodied, most have shells.
(Snails, Limpet)

Arachnids
Eight legs, two body
parts, no antennae.
(Spider, Scorpion)

Crustaceans
Mostly sea creatures.
Many legs and two
sets of antennae.
(Crab, Lobster)

Insects
Wings, six legs, three
body parts, one pair
of antennae.
(Bee, Ladybird)

Myriapods
Many legs and
body segments.
(Centipede, Millipede)

"Is a" Versus "Has a"

- A derived class demonstrates an is a relationship between it and its base class
  - Forming an "is a" relationship is one way to make a more complex class out of a simpler class

Student	is a	Person
Employee	is a	Person
Student assistent	is a	Employee
Alarm	is a	Clock
Bike	is a	Vehicle
Car	is a	Vehicle
Delivery van	is a	Car

  - Car is a more complex class compared to the more general Vehicle class

"Is a" Versus "Has a" (2)

- Another way to make a more complex class out of a simpler class is through a “has a” relationship
  - This type of relationship, called composition, occurs when a class contains an attribute of a class type
  - The Employee class contains an attribute, hireDate, of the class Date, so therefore, an Employee has a Date

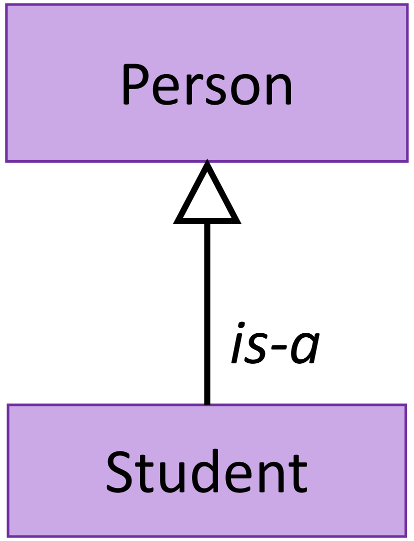
Person	has a	Name
Person	has a	Passport number
Car	has a	License number
Bike	has a	Bell
Student	has a	Study

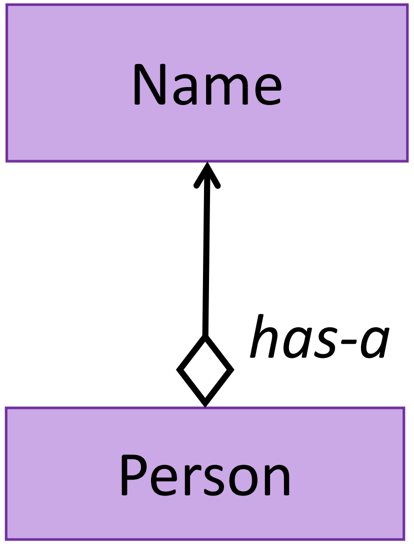
"Is a" Versus "Has a" (3)

- Both kinds of relationships are commonly used to create complex classes, often within the same class
  - Since HourlyEmployee is a derived class of Employee, and contains an instance variable of class Date, then HourlyEmployee *is an* Employee and *has a* Date

- Only a person can determine whether the right relationship is being used.

"Is a" Versus "Has a"





```
class Person {}
class Student extends Person {}
```

```
class Person {
  private Name myName;
}
```

OOP rule 2: Favor composition over inheritance

Example – base class

```
public class Person {
    private String  myName;
    private int     mySSN;

    public Person( String name, int ssn ) {
        this.myName = name;
        this.mySSN  = ssn;
    }

    @Override
    public String toString() {
        return String.format( "%s, SSN:%d", myName, mySSN );
    }
}
```

Example – derived class

```
public class Student extends Person
{
    public static enum Study{ CS, AI, MA };

    private Study myStudy;
    private int   myStudentNumber;

    public Student( String name, int ssn, Study study, int studentNr ) {
        super ( name, ssn );
        this.myStudy          = study;
        this.myStudentNumber  = studentNr;
    }

    @Override
    public String toString() {
        return String.format( "%s,%s,%d", super.toString(), myStudy, myStudentNumber );
    }
}
```

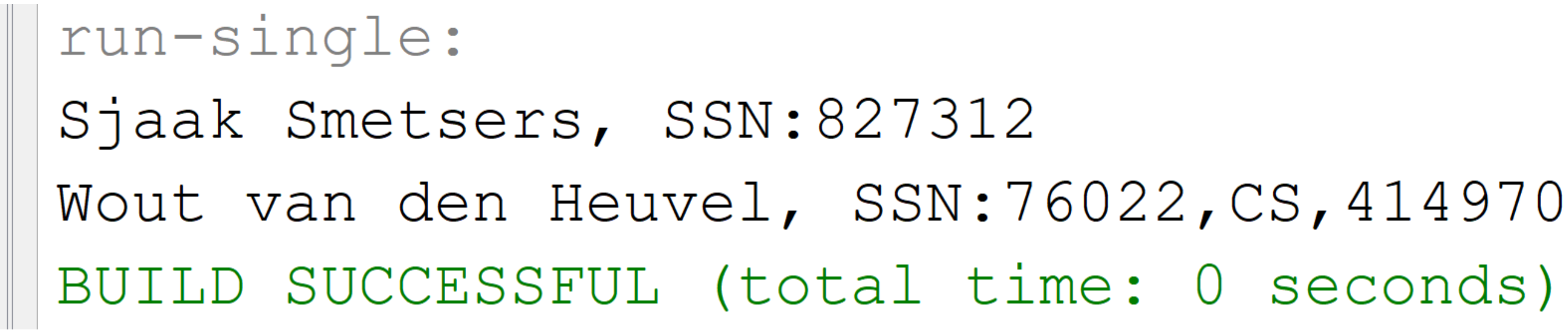
Person constructor

toString of Person

Example – main method

```
public class Week4Test {


    public static void main( String[] args ) {
        Person sjaak = new Person( "Sjaak Smetsers", 827312 );
        System.out.println( sjaak );
        Student wout = new Student( "Wout van den Heuvel", 76022, Student.Study.CS, 414970 );
        System.out.println( wout );
    }
}
```



```
run-single:
Sjaak Smetsers, SSN:827312
Wout van den Heuvel, SSN:76022,CS,414970
BUILD SUCCESSFUL (total time: 0 seconds)
```

Subtyping

- Subtyping: a Derived class is a subtype of the Base class.
- Subtyping rule

  If a *Base object* is demanded it is safe to offer a *Derived object*.

  – Example: if a `Person` is needed you can give a `Student`. Remember that everything that can be done with a `Person` can also be done with a `Student`.

`Protected` Access

- If a method or instance variable is modified by protected (rather than public or private), then it can be accessed by name
  - *Inside its own class definition*
  - *Inside any class derived from it*
- The protected modifier provides weak protection compared to the private modifier
  - *It allows direct access to any programmer who defines a suitable derived class*
  - *Therefore, instance variables should normally not be marked protected*

Overloading

- Overloading is when two or more methods in the same class have the same method name
- To be valid, any two definitions of the method name must have different *signatures*
  - *A* signature *consists of:*
    - name of a method
    - parameter list
  - Different signatures for the methods with the same name must have different numbers and/or types of parameters

Polymorphism (1)

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - *Encapsulation*
  - *Inheritance/realisation*
  - *Polymorphism*
- Polymorphism is the ability to associate many meanings to one method signature
  - *It does this through a special mechanism known as late binding or dynamic binding*
- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

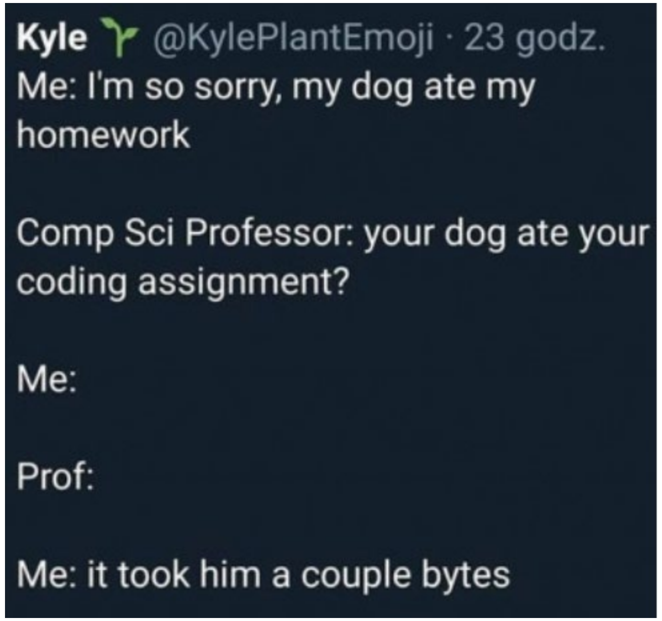
Late Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called early binding or static binding
  - Based on static types
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding
  - Based on dynamic types

Late Binding in Java

- Java uses late binding for all methods (except private, final, and static methods)
- How do I get more meanings/different functionality?
  - If a class inherits a method from its superclass, then there is the possibility to override the method.
  - Benefit of overriding: ability to define behavior that's specific to the subclass type
    - a subclass can re-implement a parent class method based on its requirement.
  - In object-oriented terms, overriding means to replace the functionality of an existing method.

Joke of the week



Kyle @KylePlantEmoji · 23 godz.

Me: I'm so sorry, my dog ate my homework

Comp Sci Professor: your dog ate your coding assignment?

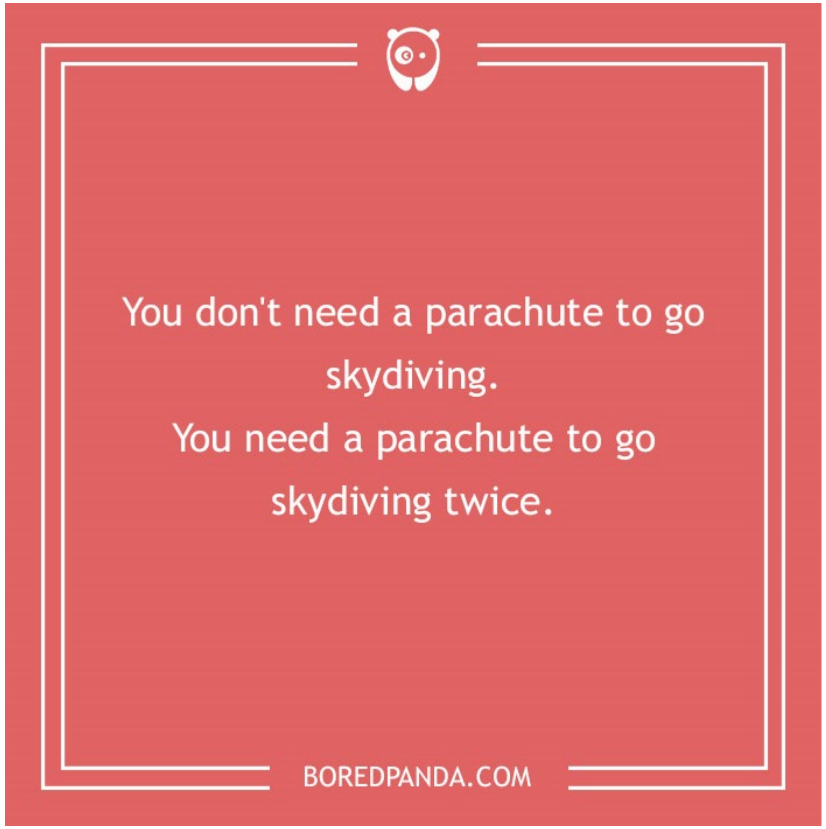
Me:

Prof:

Me: it took him a couple bytes

When you help someone fix their code but you can't fix your own

I guide others to a treasure that I cannot possess



You don't need a parachute to go skydiving.
You need a parachute to go skydiving twice.

BOREDPANDA.COM

Example --- Base class

```
public class Vehicle
{
   @Override
   public String toString( ) {
        return "A vehicle";
   }

   public final void show( ) {
        System.out.println( this.toString( ) );
   }
};
```

final: cannot be overridden

```
Vehicle v = new Vehicle ()
v.show ()
```

Prints the following on the screen

```
A vehicle
```

Derived class MotorVehicle

```
public class MotorVehicle extends Vehicle {
    private String myLicenseNr;

    public MotorVehicle( String ln ) {
        myLicenseNr = ln;
    }

    public String getLicenseNr() {
        return myLicenseNr;
    }

    @Override
    public String toString() {
        return "A motor vehicle with reg. no. " + myLicenseNr;
    }
}
```

overrides toString from Vehicle

Derived class Car

```
public class Car extends MotorVehicle
{
   public Car( String ln ) {
        super ( ln );
   }
}
```

Constructor of MotorVehicle class

Derived class Bus

```java
public class Bus extends MotorVehicle {
   private int myNrOfSeats;

   public Bus( String ln, int s ) {
      super ( ln );
      myNrOfSeats = s;
   }

   @Override
   public String toString() {
      return String.format("A bus with reg. no. %s and %d seats", getLicenseNr(), myNrOfSeats);
   }
}
```

Derived class Bike

```
public class Bike extends Vehicle {
    private int myNrofGears;

    public Bike() {
        myNrofGears = 1;
    }

    public Bike( int g ) {
        myNrofGears = g;
    }

    @Override
    public String toString() {
        return String.format( "A bike with %d gears", myNrofGears );
    }
}
```

Example – main

```
public class TestVehicle
{
   public static void main( String[] args ) {
        Car car          = new Car ( "LS-HL-97" );
        MotorVehicle mv = new MotorVehicle ( "RV-ZJ-42" );
        Bike bike        = new Bike ( 7 );
        Bus bus          = new Bus ( "AB-CD-01", 40 );

        car.show ();
        mv.show  ();
        bike.show();
        bus.show ();
   }
}
```

recall that show is defined in Vehicle only

```
Output:
A motor vehicle with reg. no. LS-HL-97
A motor vehicle with reg. no. RV-ZJ-42
A bike with 7 gears
A bus with reg. no. AB-CD-01 and 40 seats
```

Example – class Dealer

```
public class Dealer
{
   static private class Item {
      private Vehicle vehicle;
      private double  price;

      private Item( Vehicle v, double p ) {
         this.vehicle     = v;
         this.price       = p;
      }

      public String toString() {
         return String.format( "%s, price $%.2f\n", vehicle, price );
      }
  }
```

Example – class Dealer

```
    private List<Item> stock;

    public Dealer() {
          stock = new ArrayList<>();
    }

    public void buy( Vehicle v, double p ) {
          stock.add( new Item (v,p) );
    }

    public void showStock() {
          for ( Item it : stock ) {
                System.out.println( it );
          }
    }
}
```

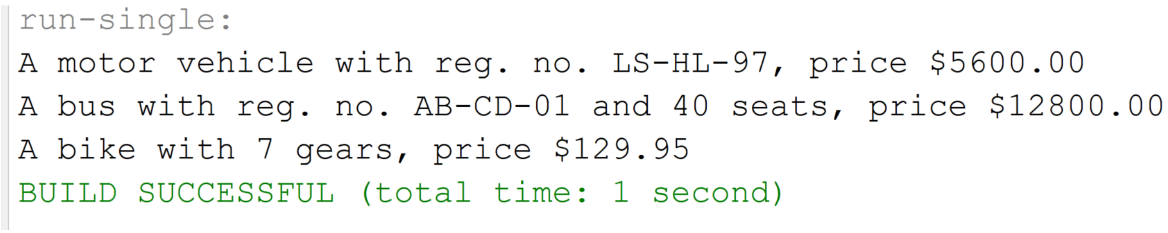
Flexible array

```
Car car         = new Car( "LS-HL-97" );
Bike bike       = new Bike( 7 );
Bus bus         = new Bus( "AB-CD-01", 40 );

Dealer dealer = new Dealer();

dealer.buy( car, 5600 );
dealer.buy( bus, 12800 );
dealer.buy( bike, 129.95 );

dealer.showStock();
```



```
run-single:
A motor vehicle with reg. no. LS-HL-97, price $5600.00
A bus with reg. no. AB-CD-01 and 40 seats, price $12800.00
A bike with 7 gears, price $129.95
BUILD SUCCESSFUL (total time: 1 second)
```

The Class `Object`

- In Java, every class is a descendent of the class `Object`
  - Every class has `Object` as its ancestor
  - Every object of every class is of type `Object`, as well as being of the type of its own class
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class `Object`
- Having an `Object` class enables methods to be written with a parameter of type `Object`
  - For example, some library methods accept an argument of type `Object` so they can be used with an argument that is an object of any class

The Class `Object`

- The class `Object` has some methods that every Java class inherits
  - For example, the `equals` and `toString` methods
- These inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

The Right Way to Define `equals`

- Since the `equals` method is always inherited from the class `Object`, methods like the following simply *overload* it:

```
public boolean equals( Employee otherEmployee ){
    … }
```

- However, this method should be *overridden*, not just overloaded:

```
public boolean equals( Object otherObject ){
    … }
```

The right equals method for Employee

typecast

```
public boolean equals( Object otherObject ) {
  if ( otherObject == null || getClass () != otherObject.getClass() ) {
    return false;
  } else {
    Employee other = (Employee) otherObject;
    return name.equals( other.name ) && hireDate.equals( other.hireDate );
  }
}
```

- Alternatively:

```
public boolean equals( Object otherObject ) {
  if ( otherObject instanceof Employee ) {
      Employee other = (Employee) otherObject ;
      return name.equals( other.name ) && hireDate.equals( other.hireDate );
  } else {
      return false;
  }
}
```

The getClass Method (and the class Class)

- Every object inherits the getClass method from the Object class
  - This method is marked final, so it cannot be overridden
- An invocation of getClass on an object returns a representation of the class (an object of class Class) that was used with **new** to create the object
  - The results of any two such invocations can be compared with **==** or **!=** to determine whether or not they represent the exact same class

```
object1.getClass() == object2.getClass()
```

Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method is called an abstract class

Abstract Class

```
public abstract class Player
{
    private String name;

    public Player( String name ) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void play( Pot p );
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods.
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract too, and must add abstract to its modifier.
- An abstract class cannot be instantiated
- A class that has no abstract methods is called a concrete class.

Exceptions

- When a program runs into a runtime error, the program terminates abnormally.
- 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 (example)

- Consider the following method

```java
private void run() {
  Scanner scan = new Scanner( "42 is the answer" );
  int i = scan.nextInt();
  System.out.println("\"" + i + "\"");
}
```

- but what if the input does not start with an integer

  like `"The answer is 42"`

- Answer: Java throws an exception

```
Exception in thread "main" java.util.InputMismatchException
```

- Program cannot proceed normally: (nested) method calls are quit.

Exceptions (more examples)

```
Object obj = null;
System.out.println( obj.toString() );
```

-
  ```
  Exception in thread "main" java.lang.NullPointerException
  ```

```
int i = 4711/0;
System.out.println( i );
```

-
  ```
  Exception in thread "main" java.lang.ArithmeticException: / by zero
  ```

```
double d = 4711.0/0;
System.out.println( d );
```

-
  ```
  Infinity
  ```

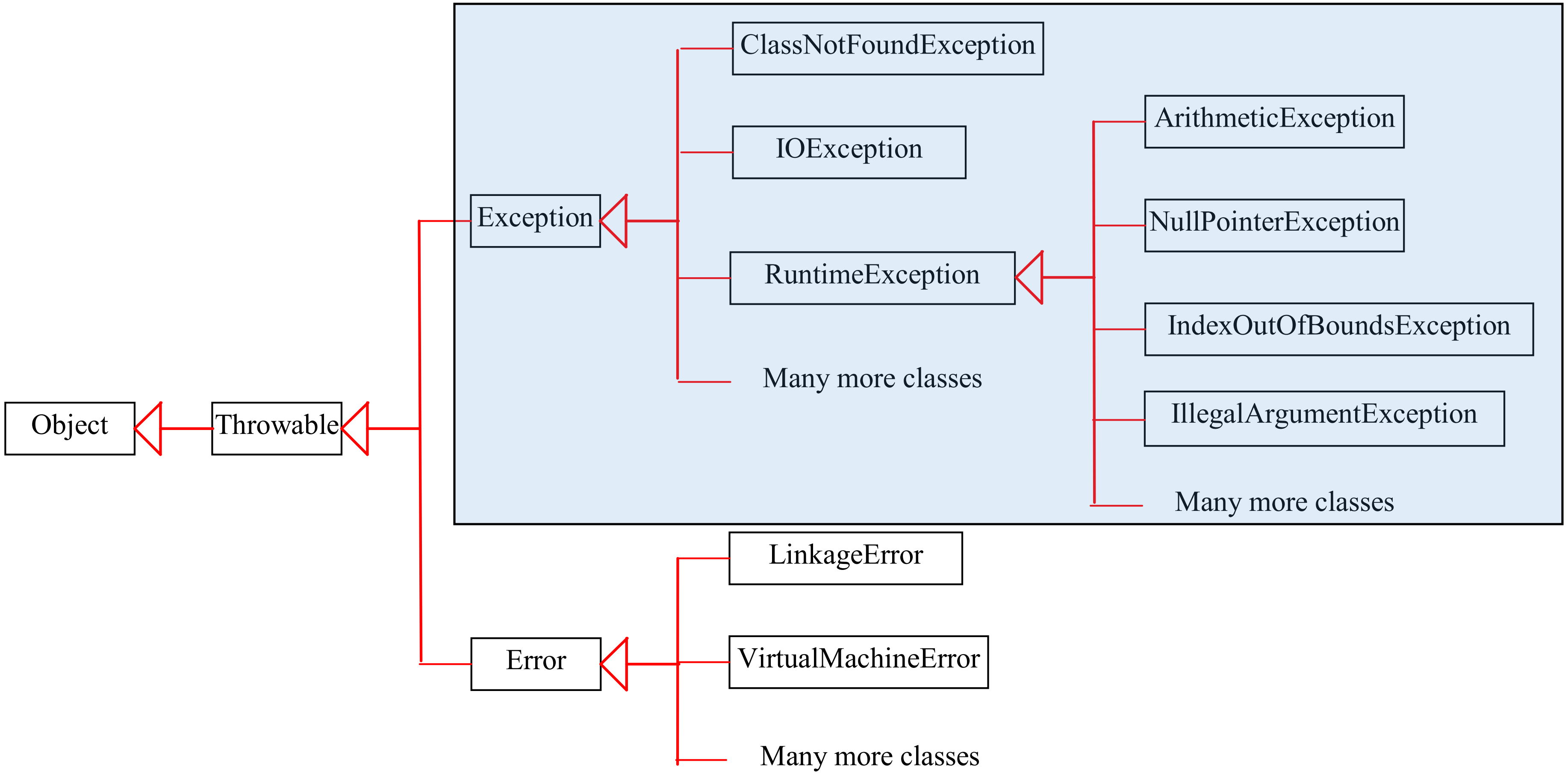
```
int[] a = { 1, 2, 3 };
System.out.println( a[3] );
```

-
  ```
  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
  ```

What is an exception?

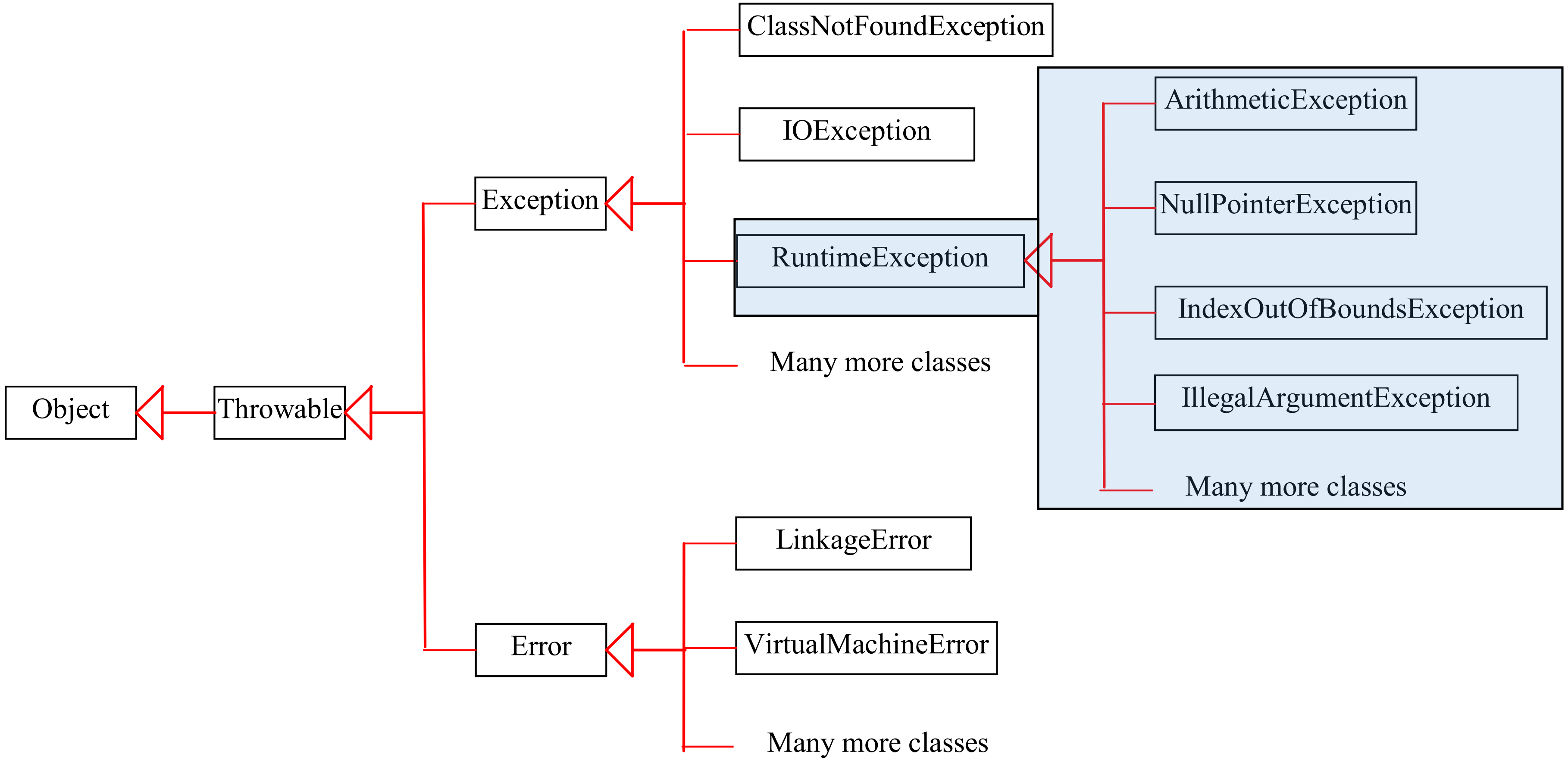
- In Java runtime errors are thrown as exceptions
- An exception is an object that represents the kind of error.
- Java provides standard exception classes, such as
  - `Exception` (checked)
  - `RuntimeException` (unchecked)
- You can use these to generate exceptions yourself or to introduce new classes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

Exception class hierarchy



Runtime Exceptions

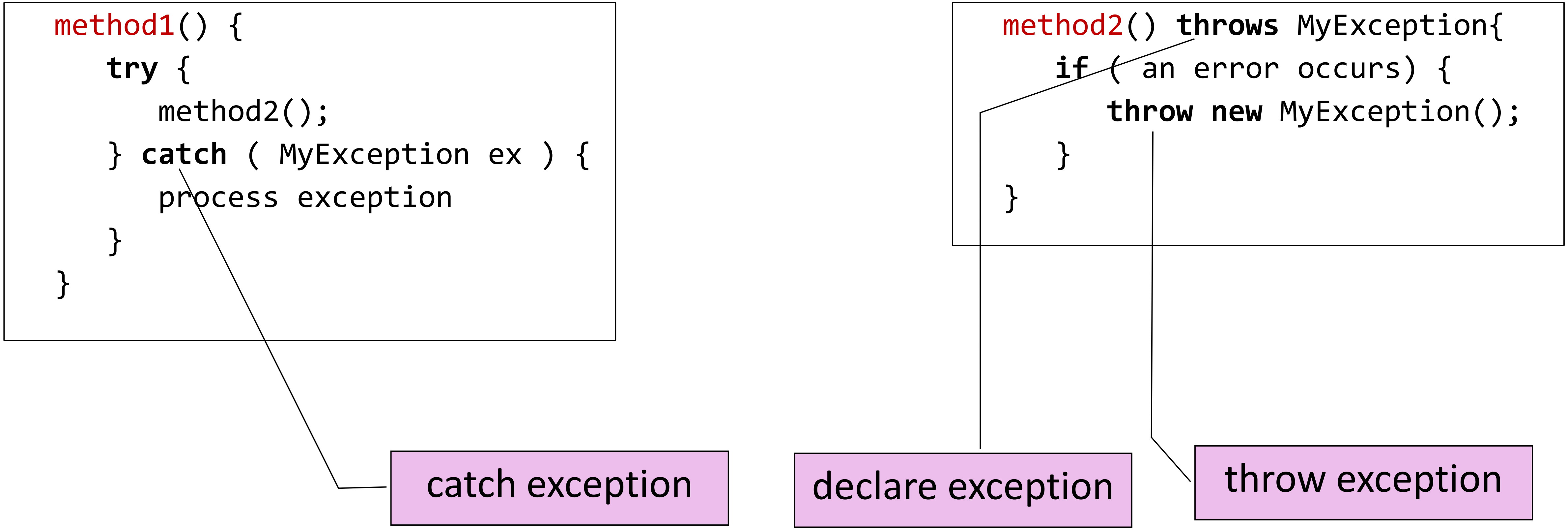
- RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.



Checked Exceptions vs. Unchecked Exceptions

- `RuntimeException`, `Error` and their subclasses are known as unchecked exceptions.
- All other exceptions are known as checked exceptions, meaning that the compiler *forces the programmer to check and deal with the exceptions*.

Declaring, Throwing, and Catching Exceptions



```
method1() {
   try {
      method2();
   } catch ( MyException ex ) {
      process exception
   }
}
```

```
method2() throws MyException{
   if ( an error occurs) {
      throw new MyException();
   }
}
```

catch exception

declare exception

throw exception

Declaring Exceptions

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

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

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.
- Here is an example

```
throw new TheException();

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

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;
}
```

Catch or Declare Checked Exceptions

- Suppose p2 is defined as follows:

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

- Java forces you to deal with checked exceptions.
  - If a method declares a checked exception, you must invoke it in a try-catch block or declare to throw the exception in the calling method.
  - For example, suppose that method p1 invokes method p2, you have to write the code as shown in (a) or (b).

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

(a)

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

(b)

Radboud University

Finally