# CSE 1325: Object-Oriented Programming

## Lecture 07

# Inheritance


## Mr. George F. Rice

george.rice@uta.edu


**Office Hours:**
**Prof Rice 12:30 Tuesday and Thursday in ERB 336**
**For TAs see this web page**

A will is a dead giveaway.

# Today's Topics

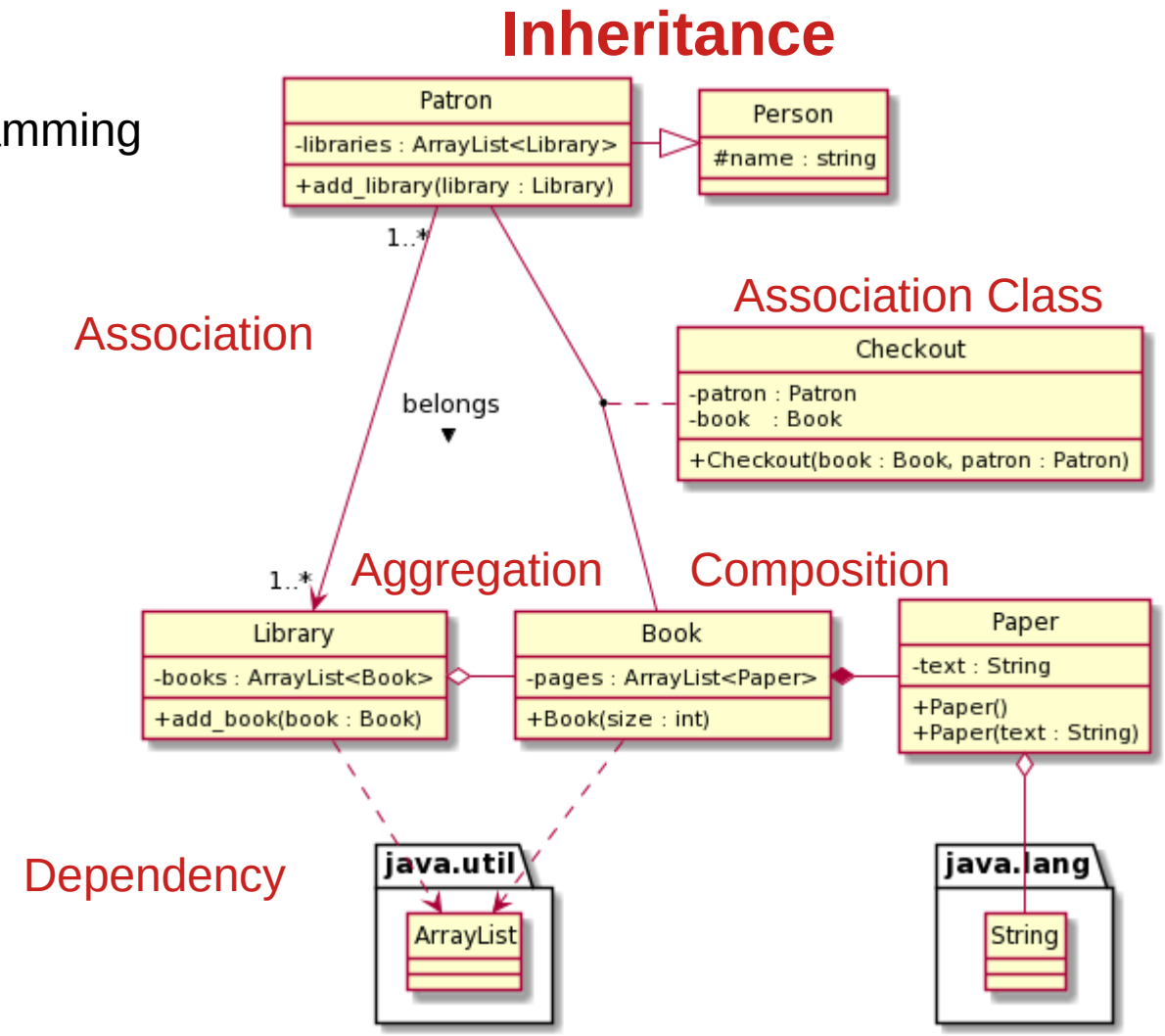
- Intro to Inheritance
  - Protected and package private class members
  - Class hierarchies
  - Implementation
  - A taste of polymorphism
  - Abstract classes and methods
  - Final classes
- Custom Exceptions
- Extending the UML

Review

# UML Relationships Summary

Object-Oriented Programming is as easy as

- Polymorphism
- **Inheritance**
- Encapsulation




**This URL generates the above diagram:**

http://www.plantuml.com/plantuml/uml/PP9Dxjem4CNtFiM8RiA7etQBgWZAgX8B90vGJH8hk77io7QgKihT6u-TDF3V63F-pPit3mxEe_L3fvXhfUxHOWULGkUEtbjP3bvyhuo-oZy2FhERh0LKqbPAC4OKd6LfqTxXmO13QKphO0z7Q_5-biv__JPM2W06My2w_X60B1mZ59xMx9c4w6jKwR4HhoeNx8MDCiHbBlZPWzIU54waS17M6HqqFe76BW3EMwcR_qCkw6r4E2XoBjz6fNqMdbMiPEbpf1EpC7-wYQerWm_Aj06DsQTvLAEj8UtD0BPpqpAGtNllOWYbGDwz-Er7uXgtWoszPElvzI9E2ZrMBJO2Vk8lp1NgXfn6tOXHuVc17RnK2tOE2Xu0KqDATVKbppShuBoOhkSvib8eeni7nGJplqNwmTX46Hwp2GTiUDoNPR5X5_ylaETpRX7CysBXQX9xBRCOdU5yokmWT3rqwXy0

# Review
# Inheritance
## with People

- When you inherit from an ancestor, you acquire (many of) their assets.
  - Some may be redirected by a will
- When a class inherits from an ancestor class, it acquires (many of) its methods and fields (also called attributes).
  - Some may be redirected by keyword directives


"Assets"

The Heir

The Ancestor

```
class You extends Rich_Aunt_Aala {
```

The "Will"

# Review
# Inheritance
## with Classes

- **Inheritance** – Reuse and extension of fields and method implementations from another class



- The original class is called the **superclass** (e.g., Exception)
- The extended class is called the **subclass** (e.g., BadArea)

"Assets"

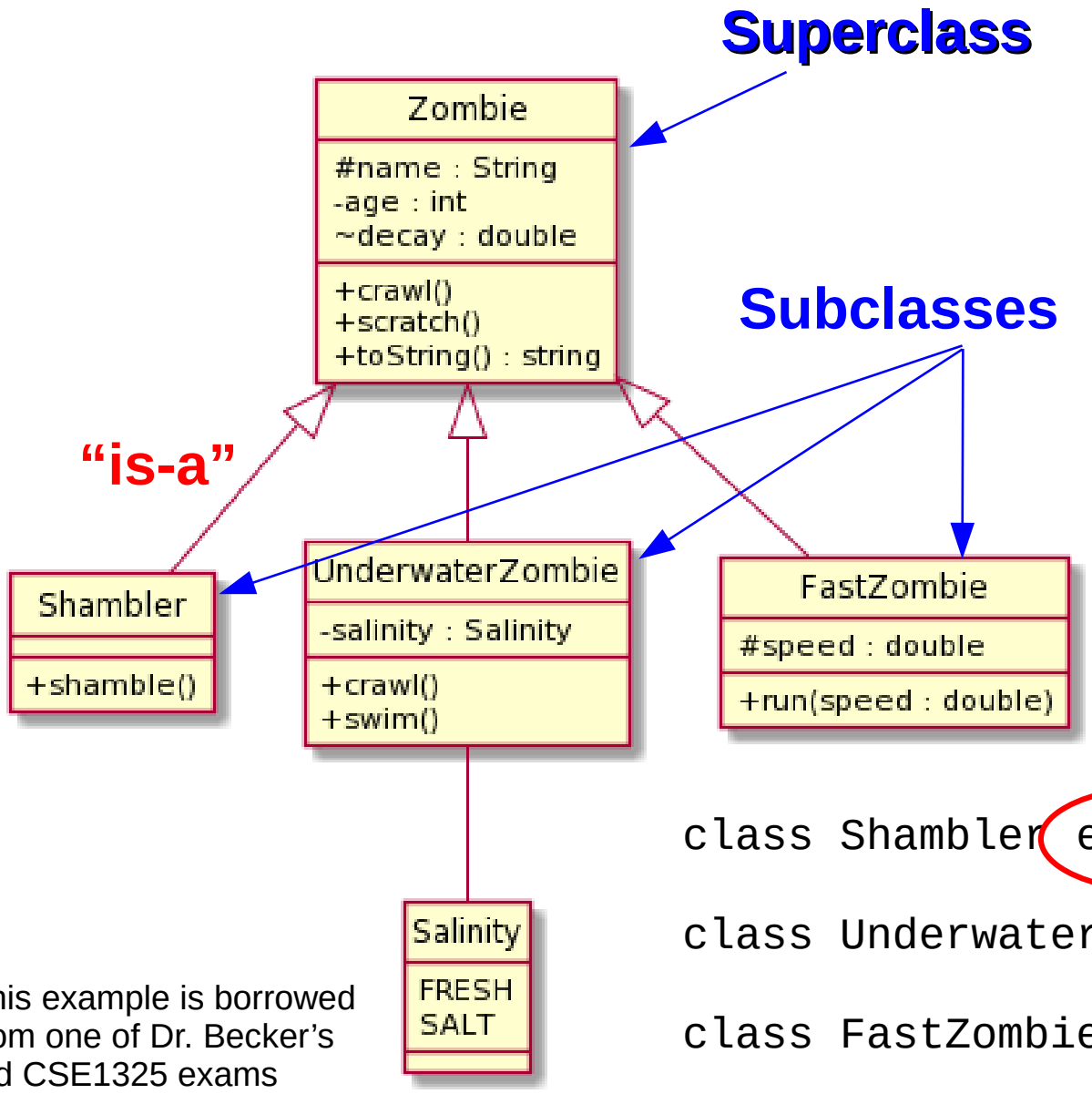
The Heir — The Ancestor

```
class BadArea extends Exception {
```

**Subclass** — **Superclass**

The "Will"

# Terminology




This example is borrowed from one of Dr. Becker's old CSE1325 exams

```
class Zombie {
```

**All 4 classes can access name, decay, crawl(), scratch(), and toString().**

**ONLY Zombie can access age (because it's private).**

**Zombie.crawl() is also used by Shambler and FastZombie(), but UnderwaterZombie.crawl() is unique (it is "overridden").**

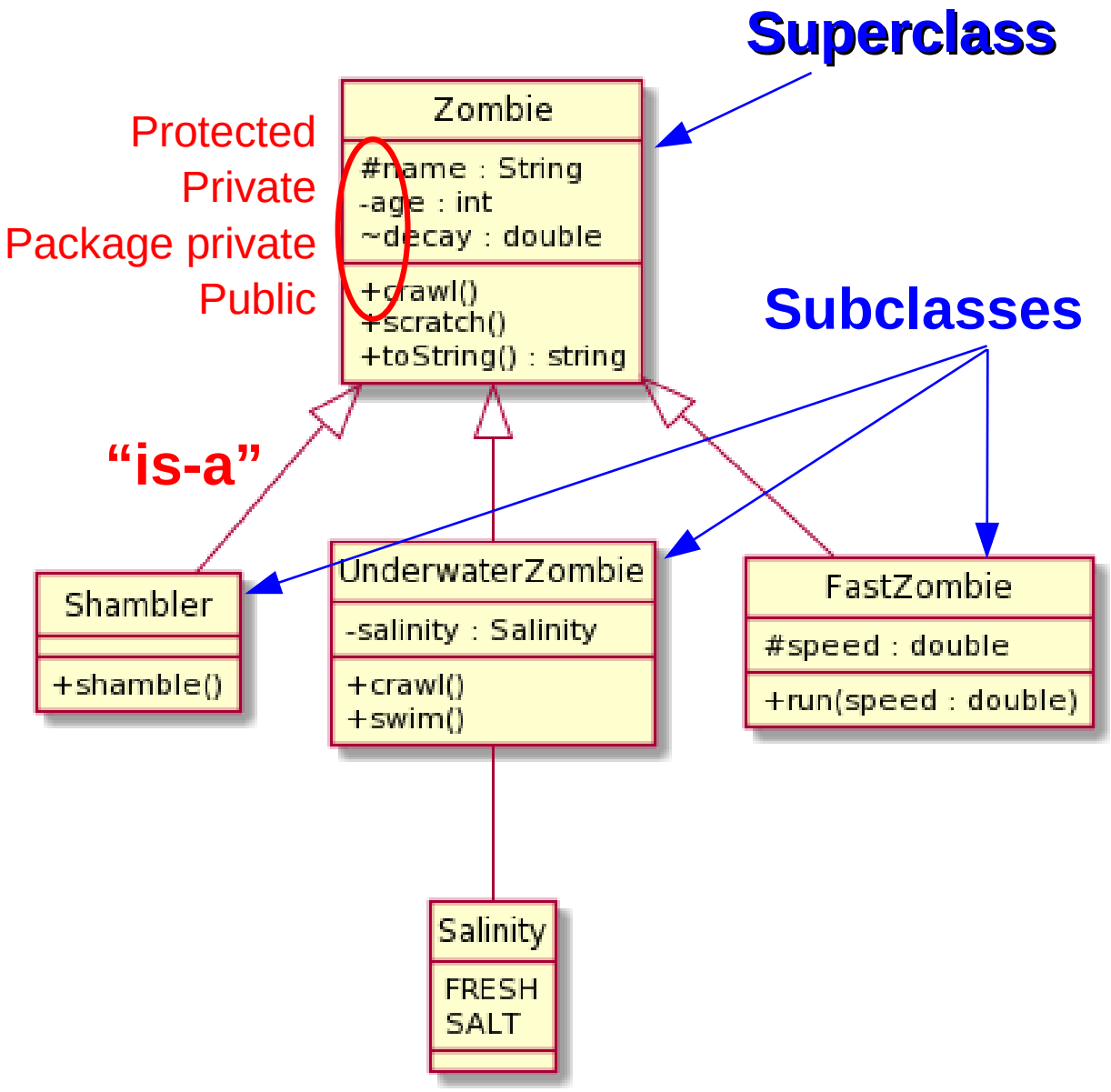
**The `extends` Java keyword declares inheritance.**

```
class Shambler extends Zombie {

class UnderwaterZombie extends Zombie {

class FastZombie extends Zombie {
```

# Protected Class Members




Making a class member public so it can be inherited also enables it to be accessed from main() et. al.

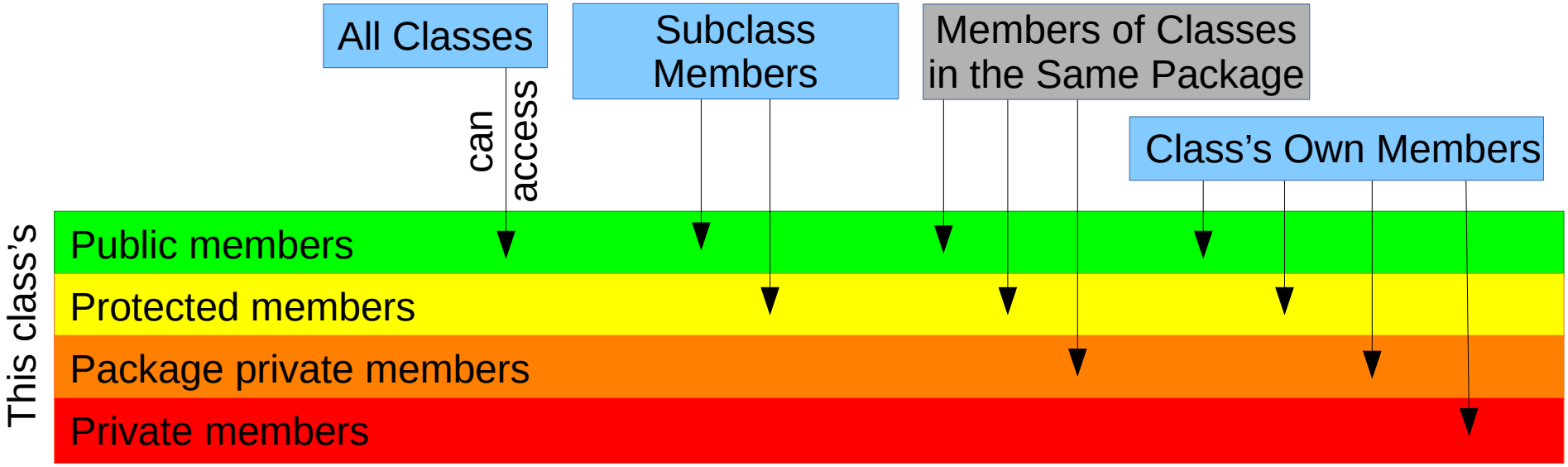
**We need a middle ground** - "only classes derived from me will have access".

We call it "**protected**", represented in the UML with '#'.

A protected member is accessible by subclasses, but is **not** accessible outside the class hierarchy.

Java also has "package private" visibility represented by '~' in the UML, which we cover next lecture.

# Java Access Model

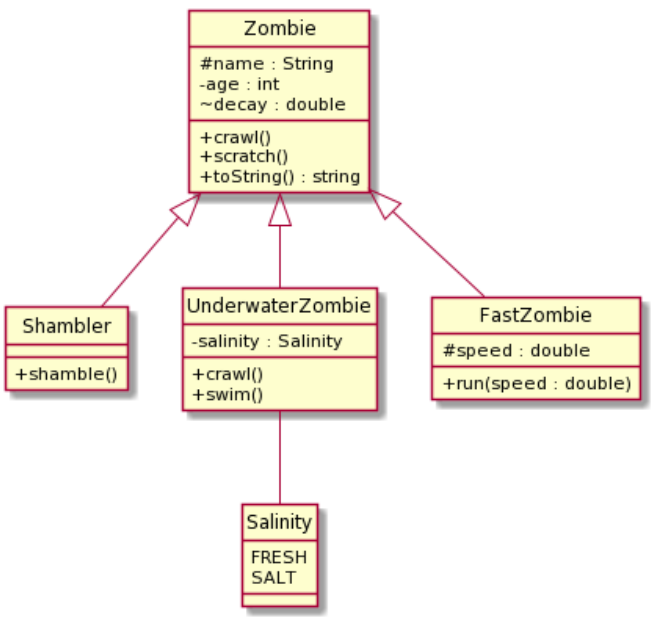


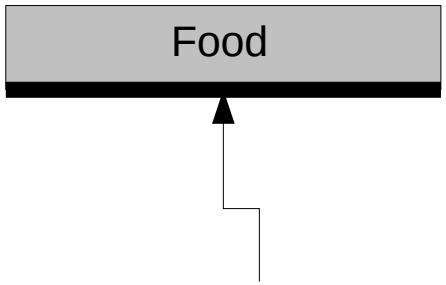

- A class or a class member (field, method, or class) can be
  - **Public** – **Anyone** can access this member
  - **Protected** – **Only class members and subclass members** can access this member*
  - **Package Private** (no modifier) **– Only class members within the same package** can access this member (we'll discuss later)
  - **Private** – **Only class members** can access this member

*Unlike most languages, Java also permits other classes in the same package to access protected members. Don't do this.

# Specify a Class Hierarchy

- A class hierarchy defines the inheritance relationships between classes
- EXERCISE: Define class hierarchies based on this superclass



# Pros and Cons of Inheritance

- **Benefits of Inheritance**
  - **Selective reuse** of superclass methods and data
  - **Reduced maintenance**, as changes to the superclass are made once and then flow automatically to the subclasses
  - **Generic interfaces**, as a call-by-reference parameter will usually accept subclass objects as well
  - **Algorithm reuse**, as algorithms written for the superclass will usually work for derived objects (e.g., sort)
- **Challenges of Inheritance**
  - **Tight coupling** of superclass and subclasses
  - **Defining the best class hierarchy** for a problem is difficult, and may be sub-optimal for other problems in the same domain
  - **Memory use** can be higher due to superclass data structure duplication

# Simple Critter in Java

- Ever visited a farm? It's noisy!
  - The noise comes from the critters who live there
  - Let's model the sounds of a barnyard

```
public class Critter {
    public Critter(int frequency) {
        this.frequency = frequency;
        this.timer = 0;
    }
    public void count() {
        if (++timer > frequency) timer = 0;
    }

    protected void say(String s) { // Prints and (attempts to) speak the parameter
    }                              // See source file for this code
    public void speak() {
        if (timer == 0) say("Generic critter sound!");
    }

    protected int frequency;
    protected int timer;
}
```

Timer is a modulo counter incremented by method count().
When 0, the critter makes a sound.
Frequency is the number of `count()` calls between sounds.

# A Generic Farm in Java

```
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.Arrays;

public class GenericFarm {
    public static void main(String[] args) {
        ArrayList<Critter> critters = new ArrayList<>();
        critters.add(new Critter(13));
        critters.add(new Critter(9));
        critters.add(new Critter(3));
        critters.add(new Critter(2));

        TimeUnit ms = TimeUnit.MILLISECONDS; // Measure “wall time” in ms

        System.out.println("W E L C O M E  T O  T H E  B A R N Y A R D !");
        for (int i=0; i<120; ++i) {
            for (Critter c: critters) {
                c.count();
                c.speak();
             }
            try {
                ms.sleep(200L + (long) (200L * Math.random()));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Let the critters speak!

This idiom pauses the program for 200-400 milliseconds, which is 0.2 to 0.4 seconds. `200L` is a long integer.

# Generic Sounds on a Farm

```
ricegf@antares:~/dev/202201/demo/barnyard$ java GenericFarm
W E L C O M E   T O   T H E   B A R N Y A R D !
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
```

# Specific Critters in Java

- But critters don't make generic sounds
  - They moo, cluck, bark, and otherwise fill the night with sound

```
public class Critter {
    public Critter(int frequency) {
        this.frequency = frequency;
        this.timer = 0;
    }
    public void count() {
        if (++timer > frequency) timer = 0;
    }

    protected void say(String s) { // Prints and (attempts to) speak the parameter
    }                              // See source file for this code
    public void speak() {
        if (timer == 0) say("Generic critter sound!");
    }

    protected int frequency;
    protected int timer;
}
```

What code needs to change to add barnyard animals (cow, chicken, dog...)?

# A Cow in Java

- NO changes to class Critter – instead, *inherit*!

```
public class Critter {
    public Critter(int frequency) {
        this.frequency = frequency;
        this.timer = 0;
    }
    public void count() {
        if (++timer > frequency) timer = 0;
    }

    protected void say(String s) { // Prints and (attempts to) speak the parameter
    }                              // See source file for this code
    public void speak() {
        if (timer == 0) say("Generic critter sound!");
    }
```

The superclass is **unchanged**. We can derive subclasses from it *independently*!

```
public class Cow extends Critter {
    public Cow(int frequency) {
        super(frequency);
    }
    @Override
    public void speak() {
        if (timer == 0) say("Moo! Mooooo!");
    }
}
```

# Simple Inheritance in Java
## (with Polymorphism for Free!)

```
class Critter {
    public Critter(int frequency) {this.frequency = frequency; timer = 0;}
    public void count() {if (++timer > frequency) timer = 0;}
    public void speak() {if (timer == 0) say("Generic critter sound!"); }

    protected int timer;
    protected int frequency;
}
```

More than one subclass, in fact!

```
class Cow extends Critter {
    public Cow(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Moo! Mooooo!"); }
}
```

```
class Chicken extends Critter {
    public Chicken(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Cluck! Cluck!"); }
}
```

```
class Dog extends Critter {
    public Dog(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Woof! Woof!"); }
}
```

# A Cow's Constructor

- NO changes to class Critter – instead, *inherit*!

```
public class Critter {
    public Critter(int frequency) {
        this.frequency = frequency;
        this.timer = 0;
    }
    public void count() {
        if (++timer > frequency) timer = 0;
    }

    protected void say(String s) { // Prints and (attempts to) speak the parameter
    }                              // See source file for this code
    public void speak() {
        if (timer == 0) say("Generic critter sound!");
    }
```

```
public class Cow extends Critter {
    public Cow(int frequency) {
        super(frequency);
    }
    @Override
    public void speak() {
        if (timer == 0) System.out
    }
}
```

However, constructors NEVER inherit. Thus, a constructor MUST be declared for each subclass (unless default is OK). To chain to the superclass constructor, we invoke super(). The **`super`** keyword represents the superclass, so **`super(frequency);`** chains to Critter's constructor.

# Simple Inheritance in Java

## (with Polymorphism for Free!)

```
class Critter {
    public Critter(int frequency) {this.frequency = frequency; timer = 0;}
    public void count() {if (++timer > frequency) timer = 0;}
    public void speak() {if (timer == 0) say("Generic critter sound!"); }

    protected int timer;
    protected int frequency;  // Rest of class omitted
```

```
class Cow extends Critter {
    public Cow(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Moo! Mooooo!"); }
}
```

Method count() is *inherited* by Cow, Chicken, and Dog – they get it "for free"!

```
class Chicken extends Critter {
    public Chicken(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Cluck! Cluck!"); }
}
```

```
class Dog extends Critter {
    public Dog(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) say("Woof! Woof!"); }
}
```

@Override is an *annotation* - javac will error if we don't override anything

Method speak() is *overridden* (replaced) by Cow, Chicken, and Dog.

# Simple Inheritance in Java
## (with Polymorphism for Free!)

Our critters ArrayList can now contain cows, dogs, and chickens! When speak() is called on a Critter from critters, Java will take care to call Cow.speak(), Dog.speak(), or Chicken.speak() depending on the type.

```
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;

class Farm {
    public static void main(String[] args) {
        ArrayList<Critter> critters = new ArrayList<>();
        critters.add(new Cow(13));    critters.add(new Dog(11));
        critters.add(new Dog(9));     critters.add(new Cow(7));
        critters.add(new Chicken(5)); critters.add(new Dog(3));
        critters.add(new Chicken(2));

        TimeUnit ms = TimeUnit.MILLISECONDS;

        System.out.println("W E L C O M E   T O   T H E   B A R N Y A R D !");
        for (int i=0; i<120; ++i) {
            for (Critter c: critters) { c.count(); c.speak(); }
            try {ms.sleep(50L);} catch (InterruptedException e) { }
        }
    }
}
```

# Simple Inheritance in Java

## (with Polymorphism for Free!)

```
ricegf@antares:~/dev/202201/demo/barnyard$ ls
build.xml     Cow.java      Dog.java   GenericFarm.java
Chicken.java  Critter.java  Farm.java
ricegf@antares:~/dev/202201/demo/barnyard$ javac Farm.java
ricegf@antares:~/dev/202201/demo/barnyard$ ls
 build.xml        Cow.java        Dog.java        GenericFarm.java
 Chicken.class    Critter.class  'Farm$1.class'
 Chicken.java     Critter.java    Farm.class
 Cow.class        Dog.class       Farm.java
ricegf@antares:~/dev/202201/demo/barnyard$ java Farm
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Moo! Mooooo!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Moo! Mooooo!
Cluck! Cluck!
Moo! Mooooo!
```

```
import j
import j

class Ma
    publ

    }
}
```

v contain

Critter
are to
(), or
on the type.

# A More Direct Initialization of an ArrayList

```
import java.util.concurrent.TimeUnit;
import java.util.ArrayList;
import java.util.Arrays;

class Farm {
    public static void main(String[] args) {
        ArrayList<Critter> critters = new ArrayList<>(
            Arrays.asList(new Cow(13),    new Dog(11), new Dog(9),  new Cow(7),
                          new Chicken(5), new Dog(3),  new Chicken(2)));

        TimeUnit ms = TimeUnit.MILLISECONDS;

        System.out.println("W E L C O M E   T O   T H E   B A R N Y A R D !");
        for (int i=0; i<120; ++i) {
            for (Critter c: critters) { c.count(); c.speak(); }
            try {ms.sleep(50L);} catch (InterruptedException e) { }
        }
    }
}
```

We'll dig more deeply into Lists in Lecture 15.

# What Happened to Critter.speak?

```
class Cow extends Critter {
    public Cow(int frequency) {super(frequency);}
    @Override
    public void speak() {
        if (timer == 0) {
            super.speak();  // This calls the superclass method speak()
            say("Moo! Mooooo!");
        }
    }
}
```

Still there – the method just needs to explicitly *ask* for it!

```
ricegf@pluto:~/dev/202008/08/java/barnyard$ java Critter
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Generic critter sound!
Moo! Mooooo!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Generic critter sound!
Moo! Mooooo!
```

# Abstract Classes and Methods

- Often, a method in a superclass can't be reasonably implemented
  - That is, the data needed isn't "known" until the subclass is defined
  - We must **ensure** that a subclass implements that method, though
  - So we make the method "abstract"
- Abstract methods make the class an **abstract class**


**Declaring a class abstract means**
**(1) we needn't – and indeed *cannot* – define A.m(),**
**(2) class A *cannot* be instanced, and**
**(3) any class derived from A *must* `@Override` and implement m() before it can be instanced.**

```
abstract class A {
    public abstract void m();
}

class B extends A {
    public void x() {System.out.println("x of B");}
    @Override
    public void m() {System.out.println("m of B");}
}

public class Good {
    public static void main(String[] args) {
        B b = new B();
        b.x();
        b.m();
    }
}
```

**B provides a definition of m() as required by A.**
**So unlike A, B *can* be instanced.**

```
ricegf@antares:~/dev/202201/demo/abstract$ javac Good.java
ricegf@antares:~/dev/202201/demo/abstract$ java Good
x of B
m of B
ricegf@antares:~/dev/202201/demo/abstract$
```

# Abstract Subclasses

- Often, a method in a class can't be reasonably implemented
  - That is, the data needed isn't "known" until the subclass is defined
  - We must **ensure** that a subclass implements that method, though
  - So we make the method "abstract"
- Abstract methods make the class an **abstract class**


**Declaring a class abstract means**
**(1) we needn't – and indeed *cannot* – define A.m(),**
**(2) A *cannot* be instanced, and**
**(3) any class derived from A *must* `@Override` and implement m() before it can be instanced.**

```
abstract class A {
    public abstract void m();
}

class B extends A {
    public void x() {System.out.println("x of B");}
    // @Override
    // public void m() {System.out.println("m of B");}
}

public class Bad {
    public static void main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

**Comment out B.m**

**We'll try to instance B, even though it didn't override m() as required by its abstract superclass. I have a bad feeling about this...**

# Abstract Methods

## (also called “Pure Virtual Functions”)

```
ricegf@pluto:~/dev/202008/08/java/abstract$ javac Bad.java
Bad.java:8: error: B is not abstract and does not override abstract method m() in A
class B extends A {
^
1 error
ricegf@pluto:~/dev/202008/08/java/abstract@
```

**Both the class AND at least one method must be declared *abstract*.**

```
abstract class A {
    public abstract void m();
}

class B extends A {
    public void x() {System.out.println("x of B");}
    // @Override
    // public void m() {System.out.println("m of B");}
}

public class Bad {
    public static void main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

**B extends A (i.e., B inherits from A), and so must `@Override` m() to be successfully instanced. If we comment out m(), Bad Things happen.**

# Correct Abstract Class

- An abstract class can ONLY be used as a superclass

**Incorrect**

```
abstract class A {
    public abstract void m();
}

class B extends A {
    public void x(){
        System.out.println("x of B");
    }
}
```

**You must *override* all abstract method(s) if you intend to instance the subclass!**

```
}

public class Bad {
    public static void
      main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

**Correct**

```
abstract class A {
    public abstract void m();
}

class B extends A {
    public void x() {
        System.out.println("x of B");
    }
    @Override
    public void m() {
        System.out.println("m of B");
    }
}

public class Good {
    public static void
      main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

# Correct Abstract Class

- An abstract class can ONLY be used as a superclass

**Incorrect**

```
abstract class A {
}

public class Bad {
    public static void
      main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

```
ricegf@antares:~/dev/202208/07/code_from_slides/abstract$ javac Bad.java
Bad.java:9: error: B is not abstract and does not override abstract method m() in A
/* abstract */ class B extends A {
               ^
1 error
ricegf@antares:~/dev/202208/07/code_from_slides/abstract@
```

You must *override* all abstract method(s)!

**Correct**

```
abstract class A {
}

class B extends A {
    @Override
    public void m() {
        System.out.println("m of B");
    }
}

public class Good {
    public static void
      main(String[] args) {
        B b = new B();
        b.x();
    }
}
```

```
ricegf@pluto:~/dev/202008/08/java/abstract$ javac Good.java
ricegf@pluto:~/dev/202008/08/java/abstract$ java Good
x of B
ricegf@pluto:~/dev/202008/08/java/abstract$
```

# Rethinking Critter as *Abstract*

```
public abstract class Critter {
    public Critter(int frequency) {
        this.frequency = frequency;
        this.timer = 0;
    }
    public void count() {
        if (++timer > frequency) timer = 0;
    }
    protected void say(String s) { /
    }                              /
    public abstract void speak();

    protected int frequenc
    protected int timer;
}
```

Remember our Barnyard? Since generic critters don't exist, we should probably make Critter an *abstract* class and speak() an *abstract* method.

```
ricegf@antares:~/dev/202201/demo/abstract$ java Farm
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Moo! Mooooo!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Woof! Woof!
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
```

**Now we can instance new cows, dogs, and chickens, but *not* generic critters – just like on a *real* farm!**

# Final Methods

- A final field cannot be *changed*
  - Other languages call this “const” or “constant”
- A final method cannot be *overridden*
  - This “locks in” the superclass implementation when it’s critical that it not change
- Consider this change to our Critters

```
abstract class Critter {
    public Critter(int frequency) {this.frequency = frequency; timer = 0;}
    public final void count() {if (++timer > frequency) timer = 0;}
    public abstract void speak();
    // Continues
```

```
class Chicken extends Critter {
    public Chicken(int frequency) {super(frequency);}
    @Override
    public void speak() {if (timer == 0) System.out.println("Cluck! Cluck!"); }
    @Override
    public final void count() {if (timer++ > frequency) timer = 0;}
}
```

# Enforcing Final Methods

- javac detects the attempt to override a final method and produces a compiler error

```
ricegf@antares:~/dev/202108/07/code_from_slides/final$ javac Critter.java
./Chicken.java:6: error: count() in Chicken cannot override count() in Critter
    public final void count() {if (timer++ > frequency) timer = 0;}
                      ^
  overridden method is final
1 error
ricegf@antares:~/dev/202108/07/code_from_slides/final@ ls
```

- Methods called from constructors should usually be final
  - This avoids weird errors if the method is overridden
- Another example (from Oracle's Tutorials) is the Chess method getFirstPlayer()
  - By the rules of chess, the white player always moves first
  - Thus this method is final to ensure subclasses don't try to change the rules of chess!

# Final Classes

- A final class works like a final method, preventing inheritance by a subclass

```
final class Dollar {
    public Dollar(int year) {this.year = year;}
    @Override
    public String toString() {return "" + year + " dollar bill";}
    protected int year;
}

public class Class {
    public static void main(String[] args) {
        Dollar dollar = new Dollar(2018);
        System.out.println(dollar);
    }
}
```

```
ricegf@antares:~/dev/202108/07/code_from_slides/final$ javac Class.java
ricegf@antares:~/dev/202108/07/code_from_slides/final$ java Class
2018 dollar bill
ricegf@antares:~/dev/202108/07/code_from_slides/final$
```

# Enforcing Final Classes

- The compiler will refuse to extend final classes

```
final class Dollar {
    public Dollar(int year) {this.year = year;}
    @Override
    public String toString() {return "" + year + " dollar bill";}
    protected int year;
}

class CounterfeitDollar extends Dollar {
    public CounterfeitDollar(int year) {super(year);}
    @Override
    public String toString() {return "" + year + " $100 dollar bill";}
}

public class Class {
    public static void main(String[] args) {
        //Dollar dollar = new Dollar(2018);
        Dollar dollar = new CounterfeitDollar(2018);
        System.out.println(dollar);
    }
```

Oops

```
ricegf@antares:~/dev/202108/07/code_from_slides/final@ javac Class.java
Class.java:8: error: cannot inherit from final Dollar
class CounterfeitDollar extends Dollar {
                                ^
1 error
ricegf@antares:~/dev/202108/07/code_from_slides/final@
```

Review

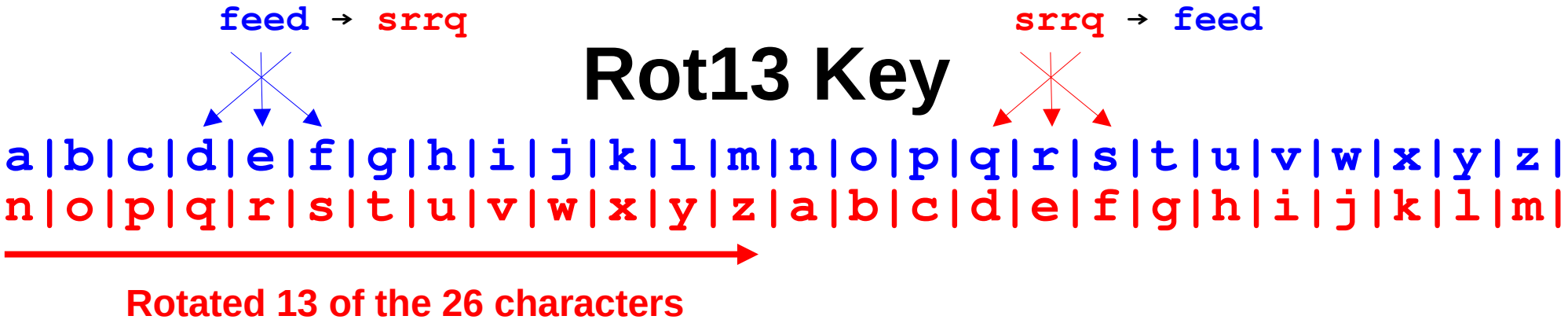
# Exceptions

- We covered predefined Exceptions in Lecture 05
  - But creating your own may be helpful
- Exception handling is general
  - You can't forget about an exception: the program will terminate if someone doesn't handle it (using a **try … catch**)
  - Any kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is **never** really simple

**Exception** – An object created to represent an error or other unusual occurrence and then propagated via special mechanisms until caught by special handling code.

# Rot13

- Rot13 (“rotate 13 character positions”) was a popular Usenet Newsgroup feature in 1970s+
  - Create key by “rotating” alphabet ahead 13 chars
  - Encode by changing blue char to red char
    - Non-alpha chars (punctuation, numbers) were simply not rotated
  - Decode is *identical* to encode! 2 Rots == original!
- Great for hiding riddle answers and such



# Throwing a RuntimeException
## (for non-Rot13 chars)

```
import java.util.Scanner;

public class Rot13 {
    static final String key = "nopqrstuvwxyzabcdefghijklm";
    public String encode(String s) {
        String result = "";
        for(char c : s.toCharArray()) {
            if(c == ' ') {result += c; continue;}
            if('a' <= c && c <= 'z') {
                result += key.charAt(c-'a');
                continue;
            }
            throw new RuntimeException("Invalid char: " + c);
        }
        return result;
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String s = in.nextLine();
        try {
            Rot13 rot13 = new Rot13();
            System.out.println(rot13.encode(s));
        } catch (RuntimeException e) {
            e.printStackTrace(); // or System.err.println(e) for less detail
        }
    }
}
```

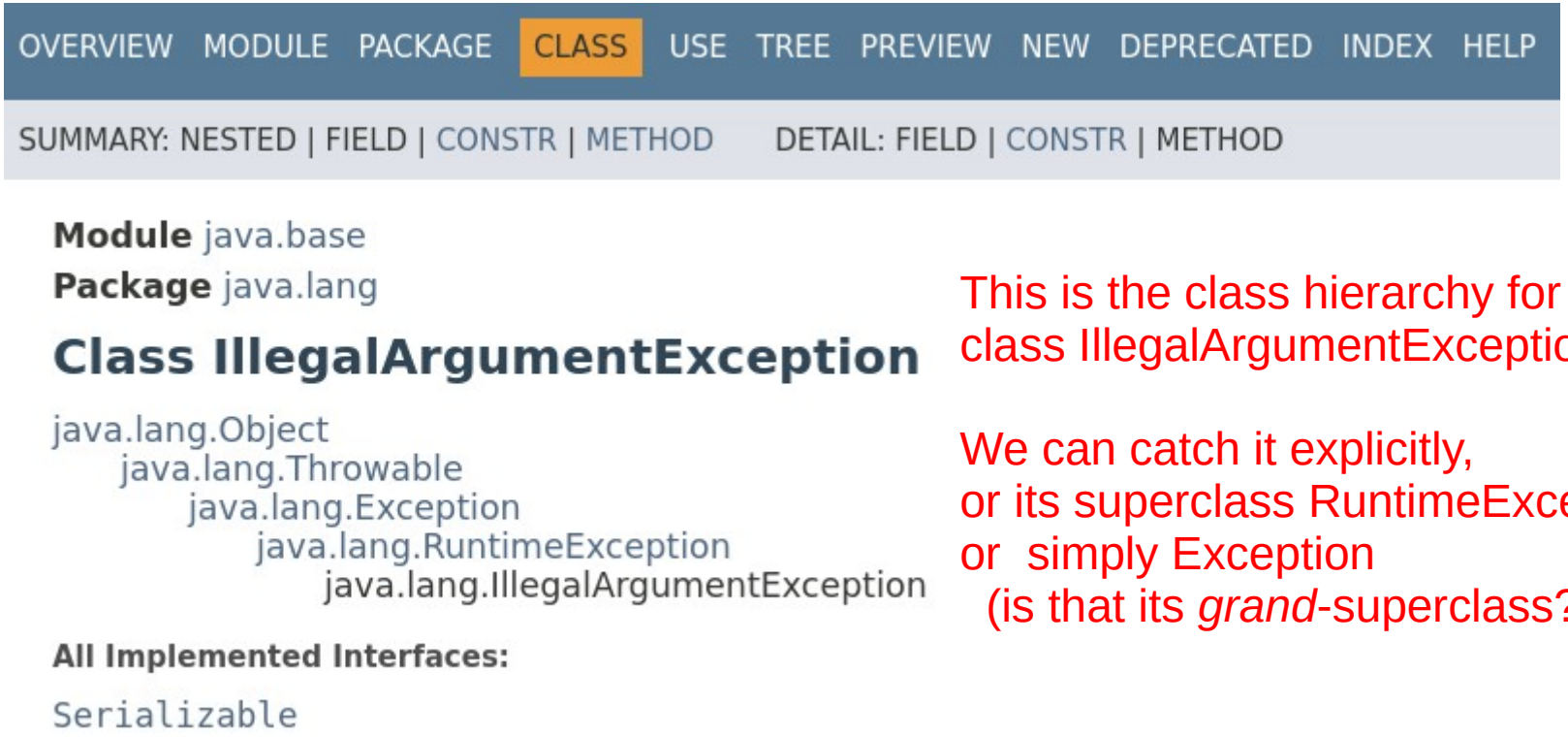
Here, instead of keeping non-alpha chars, we thrown an exception (to make a point)

# Throwing a RuntimeException
## (for non-Rot13 chars)

```
ricegf@antares:~/dev/202108/09/code_from_slides@ javac Rot13.java
ricegf@antares:~/dev/202108/09/code_from_slides$ java Rot13
Enter a string: hello world
uryyb jbeyq
ricegf@antares:~/dev/202108/09/code_from_slides$ java Rot13
Enter a string: hello world!
java.lang.RuntimeException: Invalid char: !
        at Rot13.encode(Rot13.java:13)
        at Rot13.main(Rot13.java:23)
ricegf@antares:~/dev/202108/09/code_from_slides$
```

# Defining a Custom Exception

- Custom exceptions simply inherit from Exception or one of its subclasses
  - Documentation reveals all




This is the class hierarchy for class IllegalArgumentException.

We can catch it explicitly,
or its superclass RuntimeException,
or simply Exception
(is that its *grand*-superclass?).

# Defining a Custom Exception

- We'll extend IllegalArgumentException
  - We should provide the existing constructors
    - All parameter combinations of String and Throwable
    - This may be different for other exceptions – look them up!
  - We can add whatever additional constructors, attributes, and methods that are helpful to us

Existing (chained) constructors

Custom constructor

```
public class Rot13CharException extends IllegalArgumentException {
    // Standard constructors
    public Rot13CharException() {super();}
    public Rot13CharException(String message) {super(message);}
    public Rot13CharException(Throwable err) {super(err);}
    public Rot13CharException(String message, Throwable err) {super(message,err);}

    // Custom constructor
    public Rot13CharException(String input, String encoded, char bad) {
        super("Invalid character '" + bad
            + "' at index " + encoded.length()
            + " in '" + input + "'");
    }
}
```

Simply chain to the superclass's single message constructor with our custom message!

# Throwing and Catching a Custom Exception

```
import java.util.Scanner;

public class Rot13 {
    static final String key = "nopqrstuvwxyzabcdefghijklm";
    public String encode(String s) {
        String result = "";
        for(char c : s.toCharArray()) {
            if(c == ' ') {result += c; continue;}
            if('a' <= c && c <= 'z') {
                result += key.charAt(c-'a');
                continue;
            }
            throw new Rot13CharException(s, result, c);
        }
        return result;
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a string: ");
        String s = in.nextLine();
        try {
            Rot13 rot13 = new Rot13();
            System.out.println(rot13.encode(s));
        } catch (RuntimeException e) {
            e.printStackTrace(); // or System.err.println(e) for less detail
        }
    }
}
```

No change to main(), since Rot13CharException extends IllegalArgumentException which extends RuntimeException.

So Rot13CharException “is a” RuntimeException! Inheritance!

# Viewing a Custom Exception

```
ricegf@antares:~/dev/202108/09/code_from_slides$ javac Rot13.java
ricegf@antares:~/dev/202108/09/code_from_slides$ java Rot13
Enter a string: hello world!
Rot13CharException: Invalid character '!' at index 11 in 'hello world!'
        at Rot13.encode(Rot13.java:28)
        at Rot13.main(Rot13.java:38)
ricegf@antares:~/dev/202108/09/code_from_slides$
```

Our custom exception constructs a standard, more informative message
than we could expect from a generic RuntimeException or InvalidArgumentException.

# Exception Handling Outline

- Declare an exception (`class BadArea extends Exception { … }`)
  - Optional – you can (and often should) use a pre-defined exception
    - **`RuntimeException("Bad dates")`** is a popular choice
    - **`IndexOutOfBoundsException("vector index too big")`** is also popular
- Throw an exception when an error occurs (`throw new BadArea();`)
  - Optional – many library methods already throw exceptions
- Define a scope in which to watch for an exception (`try { }`)
  - The code inside the curly braces (the "try scope") is monitored for exceptions – this is usually at a higher call level than the **`throw`**
- Immediately after the try scope, define one *or more* exception handling scopes (`catch (Exception e) { }`)
  - Add code inside the curly braces to handle each exception
- Recover if possible, rethrow, or exit as a last resort

# Exception Handling
## What should I do when I catch an exception?

- Fix the problem and continue
  - Exiting the catch scope and continuing is the default action in Java
- Re-throw the *same* exception, e.g., **`throw e;`**
  - This exits the current catch scope and looks for a matching catch in the broader (or higher) scopes
  - The original exception may be included as the Throwable parameter
- Throw a *different* exception, e.g.,
  **`throw RuntimeException{“REALLY bad exception”, e};`**

  Java cleverly retains the original exception!
  - As with a re-throw, this exits the current catch scope in search of a matching catch in broader scopes
- Abort the program
  - Print a message to System.err or display an error dialog message
  - **`System.exit`** with a non-zero result to report the error to the OS OR

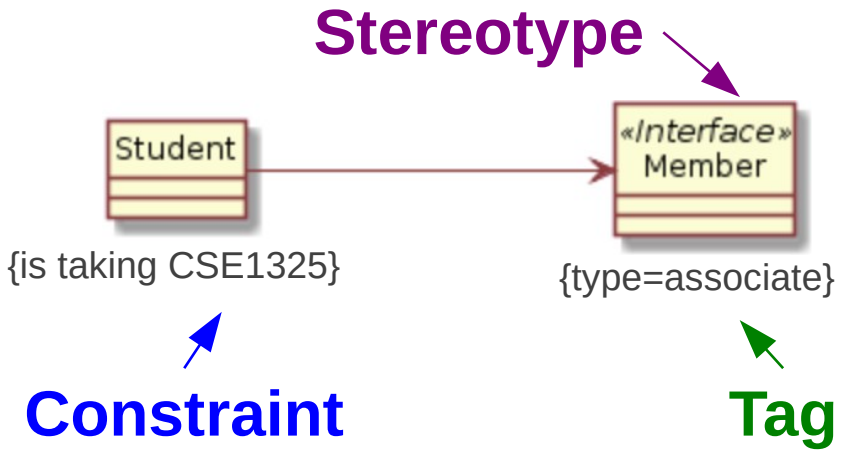
    assert something false (must run with **`java –ea *classname*`**)

# Java Offers a LOT of Exceptions

- I count close to 600
- Programming Guide seems to have a useful list
  - https://programming.guide/java/list-of-java-exceptions.html
- But add your own when it improves clarity

# Extending the UML

- UML offers 3 distinct ways to extend its notation
  - **Stereotype** – guillemets (« »)-enclosed specialization
    - In essence creates a custom element, member, or relationship type
  - **Tag** – curly-brace-enclosed assignment of value to tag name,
    - May prove useful to specify use of a class or to control code generation
  - **Constraint** – curly-brace-enclosed Boolean expression
    - Adds a condition, restriction, or assertion on the class




A given tool may specify certain semantic interpretations it will apply to specific extensions, but the UML spec does NOT limit what tags, stereotypes, or constraints you may create for your diagrams.

YOU decide what these mean!

# Example Stereotypes

- You may create any stereotype that clarifies the model, but here are some common ones I've seen (you needn't know these definitions for the exam)
- For classes
  - **«enum»** indicates an enumeration class (though this is usually obvious from the values)
  - **«interface»** indicates a Java interface (next lecture!) or a C/C++ .h file
  - **«exception»** inherits from Exception (and could alternately be shown that way)
  - **«singleton»** specifies that the class may have only one instance in the system
- For class members (follows the return type or field type)
  - **«library»** field references a static or (more commonly) a dynamic library
  - **«script»** field references a text type that can be interpreted as a program
  - **«create»** method returns a new object (for other languages, may also see «destroy»)
- For class relationships (placed near relationship line)
  - **«call»** represents a consumer class invoking a supplier class service
  - **«create»** requests a new object be instanced or «derive» requests the object be cloned
  - **«send»** represents transmission of a message or network packet

# Example Constraints and Tags

- A **constraint** is a boolean expression – English, code, or Object Constraint Language (OCL – see https://modeling-languages.com/ocl-tutorial/). Long constraints may be placed within a comment box.
  - `{gpa > 3.0}` constrains class student to a minimum GPA, e.g., for a TA
  - `{isParent}` constrains class Person to instances that have children
  - `{age >= 21}` constrains instances of class Person entering a US bar
  - `{section.size <= 60}` constrains the size of our CSE1325 section instances
  - `{isEncrypted}` (with the `«send»` stereotype) constrains messages to be secure
- **Tags** are assignments (with =) that give additional information about a class, member, or relationship
  - `{vendor = "System76"}` encodes direction for a corporate Buyer
  - `{language = Java}` specifies which language to generate from the model
  - `{region = shared}` specifies instances of the class or objects referenced by a class field be always allocated to shared memory

# What We Learned Today

- New visibility beyond public and private
  - Protected is visible to subclasses
  - Package private is visible within the package
- Class hierarchies
- Implementation using extends and @Override
- A taste of polymorphism with ArrayList
- Abstract classes and methods
  - One or more methods with no implementation
  - Abstract class may be used as superclass, variable type, and parameter type
  - Abstract class may NOT be instanced
- Final classes may not be extended
  - Final methods may not be overridden