Introduction to Java programming

Object-oriented design

Michiel Noback (NOMI)
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Introduction

- We will leave the cells and look at how to **model** sequence objects
- In this process, we will (re-)visit three very important new design tools: **interfaces**, **abstract classes** and **enums**
- First, we'll look at how to keep code well organized when our projects are becoming bigger than one or two classes, by using **packages**

PART 1: Packages & imports

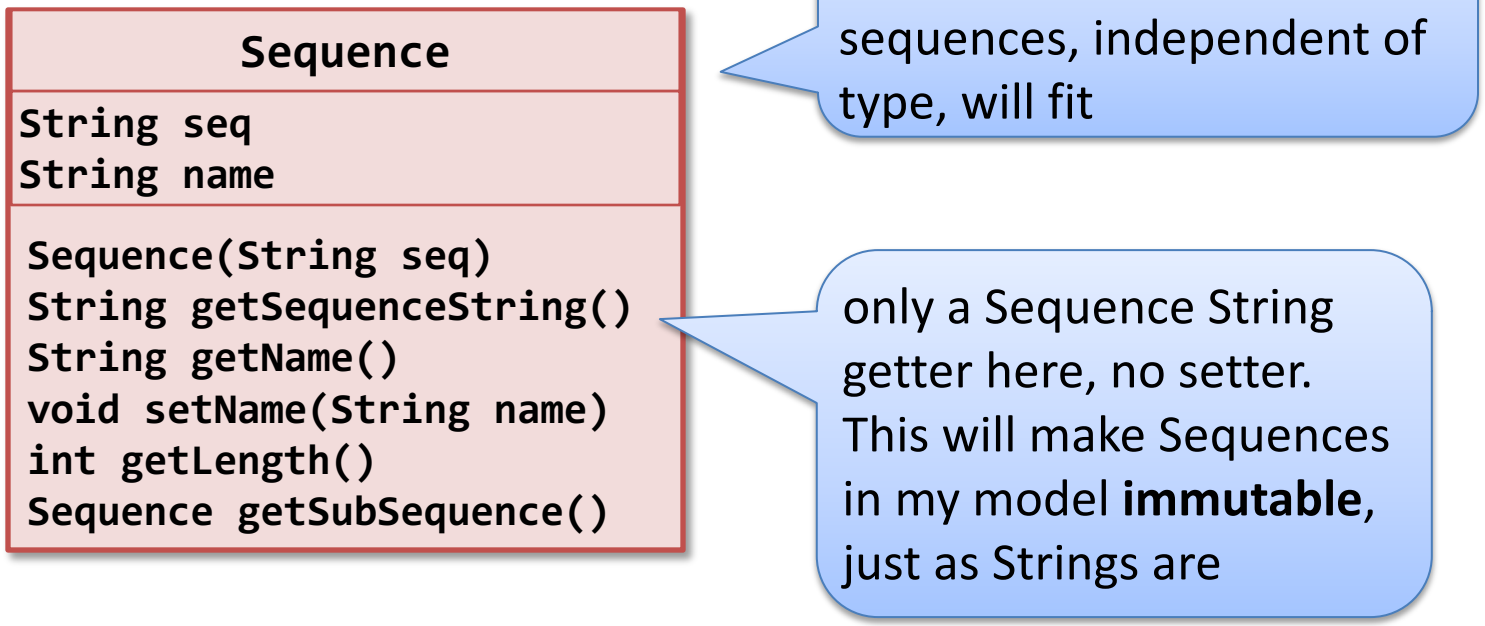
Keeping your sequences organized

- Let’s go into genetics: We are going to build a nice **data model** of biological sequences
- Take a minute to think about how ***you*** would model biological sequences such as protein and DNA

A unified sequence model

- What sequences can you think of?
- What methods and properties are shared by all biological sequences; where are differences?
- What is unique to DNA and RNA? What to protein?
- We will start here with a simple, basic model on which you can expand later

A unified sequence model



How to organize it all

- So far, you have probably stored all your class files inside a single (default) source directory
- For larger projects this will rapidly become a big unordered mess
- Besides this, you will run into **namespace** problems: how many Sequence classes do you think are out there?

Namespace

- A **namespace** is a context in

  which a class name has meaning
- For example, many Java applications have a class called **`User`**. How can the JVM know which of these classes to use if there are more to be found on the class path?
- In Java this is solved using **packages**

Packages

- All serious Java projects have their code organized into **packages**
- Packages are containers that keep related classes together and avoid conflicts with classes with identical names
- To keep packages (and thus namespaces) unique, they are usually formed by reversing the programmer/company web address:

  nl.bioinf is a package name prefix I often use

Packages

- Since there are more programmers working behind **`nl.bioinf`**, I also add my teacher abbreviation:

  `nl.bioinf.nomi`
- Finally, all sequence-related classes I put in package sequences. This gives a full package name

  `nl.bioinf.nomi.sequences`
- This is a long name, but quite essential

Packages

- Most of the core Java classes that you have worked with so far are in package **`java.lang`**
- The ***fully qualified*** name for class String is actually **`java.lang.String`**
- Since **`java.lang`** is the default package you do not have to type this prefix
- Have a look at http://docs.oracle.com/javase/7/docs/api/, find package **`java.lang`** and you will see all the familiar classes `Double, String, Math, Character` etc.

Packages

- Packages are represented in the directory structure containing the code.
- When you have a package `nl.bioinf.sequences` and this package contains a class named `Sequence`, you will **have** to place the class file of Sequence at:

…/nl/bioinf/sequences/Sequence.class

- You will usually use an IDE that takes care of these matters for you

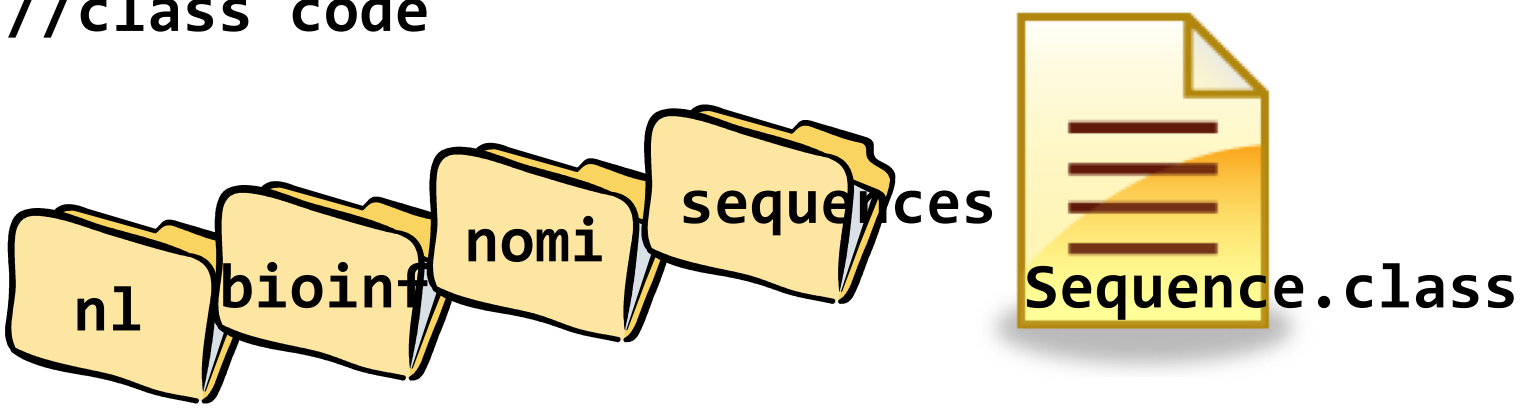
Packages and imports

- When you use classes that are not from java.lang, or the same package (=directory) your current class is in, you will have to refer to them using a **fully qualified name** or **import** them
- Let’s look at an example

Package declaration

Class Sequence is declared to be included in this package. Package declarations are placed on the first line of the source file, **outside** the actual class code

```
package nl.bioinf.nomi.sequences;

public class Sequence{
  //class code
}
```



Packages and imports

```
package nl.bioinf.nomi.sequences;

public class Sequence{

   void somemethod(){
     nl.bioinf.nomi.io.SequenceReader sr =
           new nl.bioinf.nomi.io.SequenceReader();
   }
   //more class code
}
```

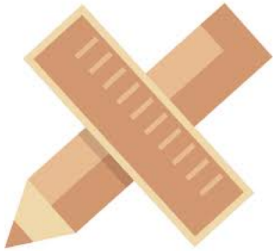
This is the long form to use a class from another package: by using the **fully qualified name**

Packages and imports

```
package nl.bioinf.nomi.sequences;

import nl.bioinf.nomi.io.SequenceReader;

public class Sequence{

  void somemethod(){
    SequenceReader sr = new SequenceReader();
  }
  //more class code
}
```

The short form: put an import statement that will make class SequenceReader available inside class Sequence

Because of the import statement, the short class name SequenceReader can now be used inside this class



Design rule

- Always put Java classes in well-defined packages with a correct name structure

PART 2: Creating a model

Back to the Sequence model: code

```
public class Sequence {
  private String seq;
  private String name;

   /*constructs with the sequence string*/
   public Sequence(String seqStr) {
     this.seq = seqStr;
   }

  /*returns the sequence string*/
  public String getSequenceString() { return seq; }
  /*setter for sequence name*/
  public void setName( String name ) { this.name = name; }
  /*getter for sequence name*/
  public String getName() { return name; }
  /*returns the length of the sequence*/
  public int getLength() { return seq.length() }
  /*returns a subSequence of the sequence*/
  public String getSubSequence() { //for you to implement }
  /*overrides Object.toString()*/
  public String toString() { //for you to override }
}
```

I made the construction of Sequence object with a sequence string mandatory. Do you agree?

Sequence model code

```
public class Sequence{
  private String seq;
  private String name;
  /*constructs with the sequence string*/
  public Sequence( String seqStr ){
    this.seq = seqStr;
  }
  /*returns the sequence string*/
  public String getSequenceString(){ return seq; }

  /*setter for sequence name*/
  public void setName(String name) { this.name = name;}
  /*getter for sequence name*/
  public String getName(){ return name; }
  /*returns the length of the sequence*/
  public int getLength() { return seq.length(); }

  /*returns a subSequence of the sequence*/
  public Sequence getSubSequence(int start, int stop) { /
  /*overrides Object.toString()*/
  public String toString() { //for you to override }
}
```

simple getter and setter for the sequence name

To implement getLength() just ***delegate*** to String

Sequence model code

```
public class Sequence{
  private String seq;
  private String name;
  /*constructs with the sequence string*/
  public Sequence(String seqStr) { this.seq = seqStr; }
  /*returns the sequence string*/
  public String getSequenceString() { return seq; }
  /*setter for sequence name*/
  public void setName(String name) { this.name = nam
  /*getter for sequence name*/
  public String getName() { return name; }
  /*returns the length of the sequence*/
  public int getLength() { return seq.length() }

   /*returns a subSequence of the sequence*/
   public Sequence getSubSequence(int start, int stop) {
     //for you to implement
   }

  /*overrides Object.toString()*/
  public String toString() { //for you to override }
}
```

Getting a subsequence from a bigger one is something you do **all the time** in bioinformatics

Sequence model code

```
public class Sequence{
  private String seq;
  private String name;
  /*constructs with the sequence string*/
  public Sequence( String seqStr ){ this.seq = seqStr; }
  /*returns the sequence string*/
  public String getSequenceString(){ return seq; }
  /*setter for sequence name*/
  public void setName( String name ){ this.n
  /*getter for sequence name*/
  public String getName(){ return name; }
  /*returns the length of the sequence*/
  public int getLength(){ return seq.length() }
  /*returns a subSequence of the sequence*/
  public Sequence getSubSequence( int start, int st    ){ //for you to implement }

   /*overrides Object.toString()*/
   public String toString(){
     //for you to override
   }
}
```

Overriding Object.toString() is very useful, especially during development and debugging

Sequence model code

```
/**
 * returns a subSequence of the sequence, starting
 * (inclusive) at position start and ending (inclusive)
 * at position stop. Sequence coordinates are 1-based,
 * not 0-based like arrays
 */
public Sequence getSubSequence(int start, int stop) {
  //for you to implement
}
/**
 * overrides Object.toString(); returns a String
 * representation of the Sequence object
 */
public String toString() {
  //for you to override
}
```

Do it now, it's not that hard!

Implementing getSubSequence()

```
public Sequence getSubSequence(int start, int stop) {
  String subStr = this.seq.substring(start-1, stop);
  Sequence subSeq = new Sequence(subStr);
  String subSeqName = this.name
                        + "@" + start
                        + "_" + stop;
  subSeq.setName(subSeqName);
  return subSeq;
}
```

Javadoc

String.substring(int beginIndex, int endIndex):
Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1.
See http://java.sun.com/javase/7/docs/api/

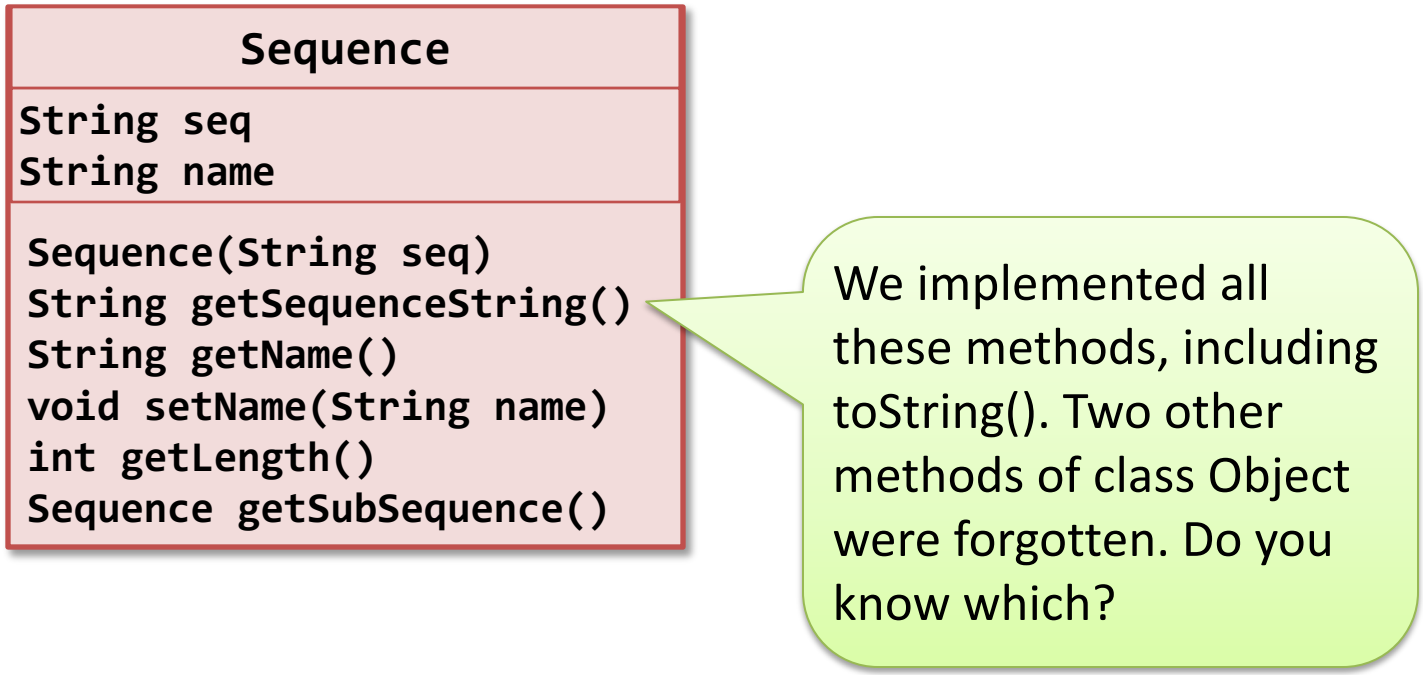
Overriding Object.toString()

```
public String toString() {
   String strRep = "Sequence[name="
             + this.name
             + " length="
             + this.getLength()
             + "]";
   return strRep;
}
```

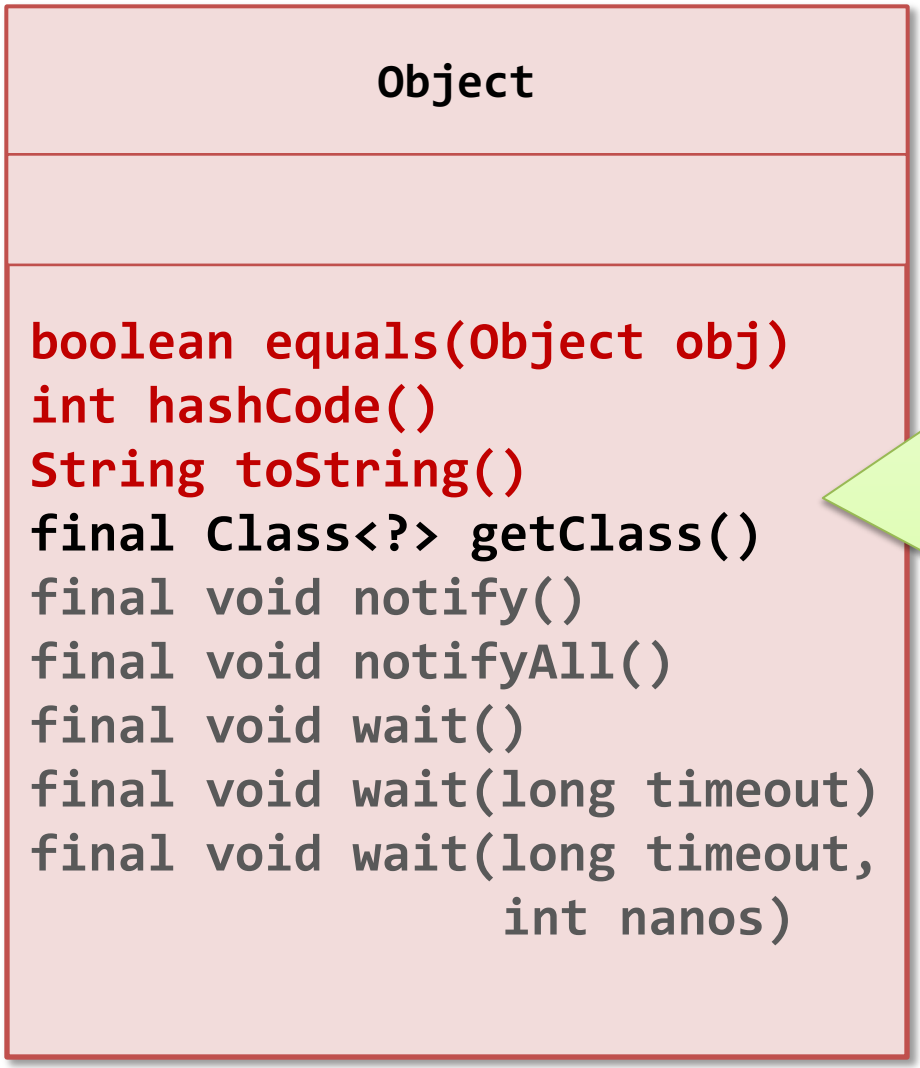
Creates a nice String representation of this sequence

By the way, whenever you do a lot of String concatenation, you should absolutely use class StringBuilder!!

Before we start testing: what did we forget



Remember class Object?



We forgot hashCode() and equals()!
Since we are going to work with many sequences, in advanced data structures and algorithms, they really need to be implemented correctly.

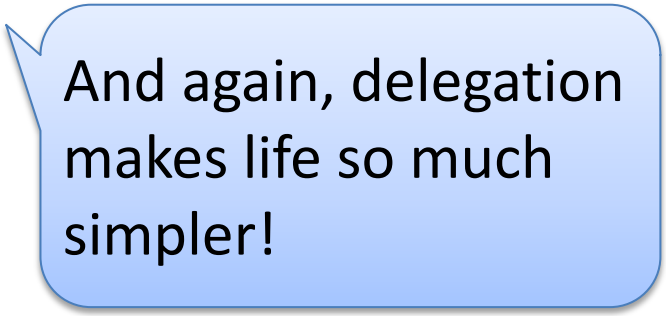
Overriding Object.equals()

```
public boolean equals(Object other) {
  return(other instanceof Sequence
      && this.seq.equals(
        (Sequence)other.getSequenceString)
      );
}
```

Delegate the equals logic to class String (again)

Overriding Object.hashCode()

```
public int hashCode(){
   return this.seq.hashCode();
}
```



Going for a test drive

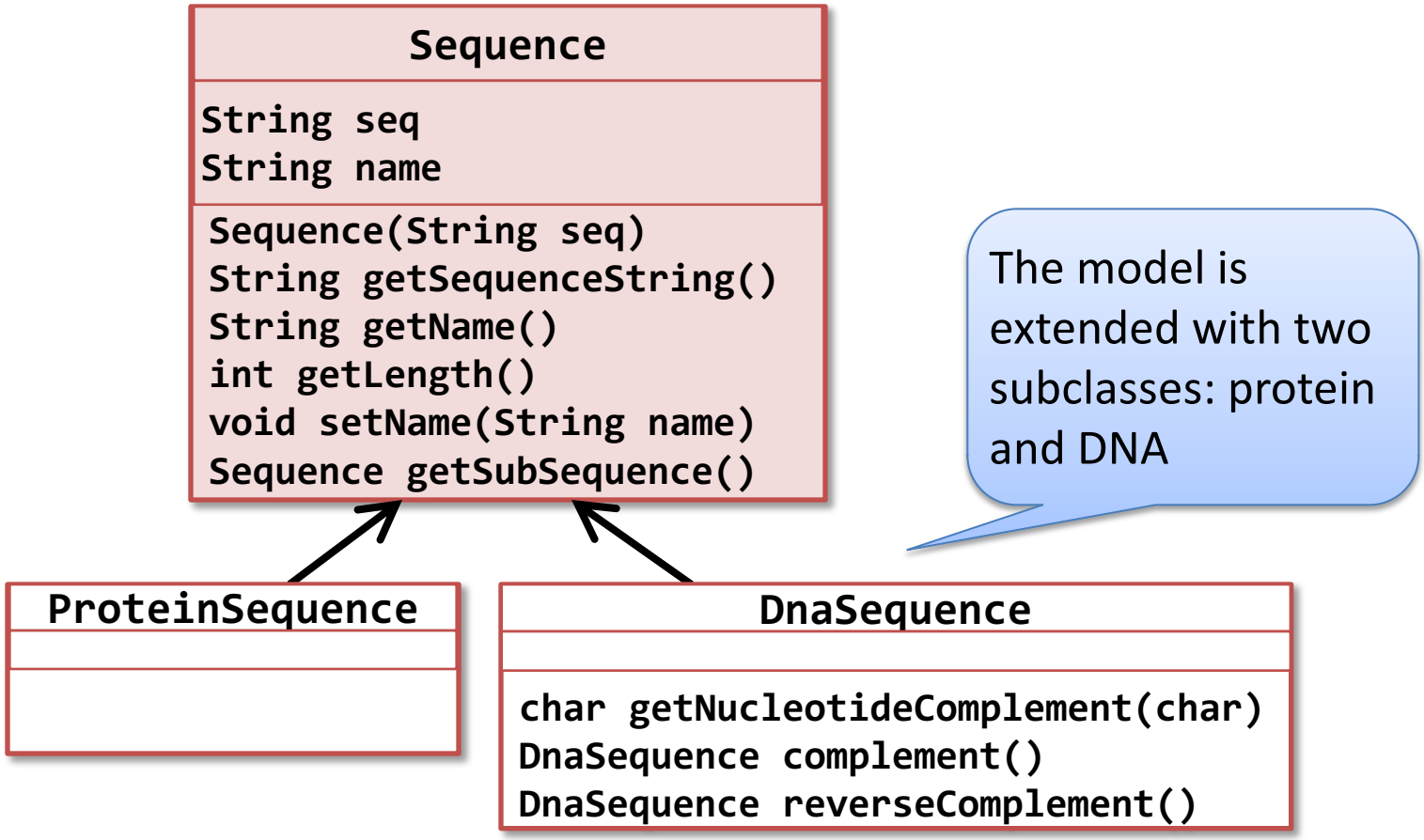
```
public class SequenceTester{
  public static void main(String[] args) {
    SequenceTester tester = new SequenceTester ();
    tester.test();
  }
  private void test() {
    Sequence seq = new Sequence("SOMERANDOMTEXT");
    seq.setName("Sequence1");
    System.out.println(seq.toString());
    System.out.println(seq.getSequenceString());
    System.out.println("creating subsequence from 2 to 6: "
              + seq.getSubSequence(2,6));
    System.out.println("printing new substring from 2 to 6: "
              + seq.getSubSequence(2,6).getSequenceString());
}
```

```
$ java SequenceTester
Sequence[name=Sequence1 length=14]
SOMERANDOMTEXT
creating subsequence from 2 to 6: Sequence[name=Sequence1@2_6 length=5]
printing new substring from 2 to 6: OMERA
```

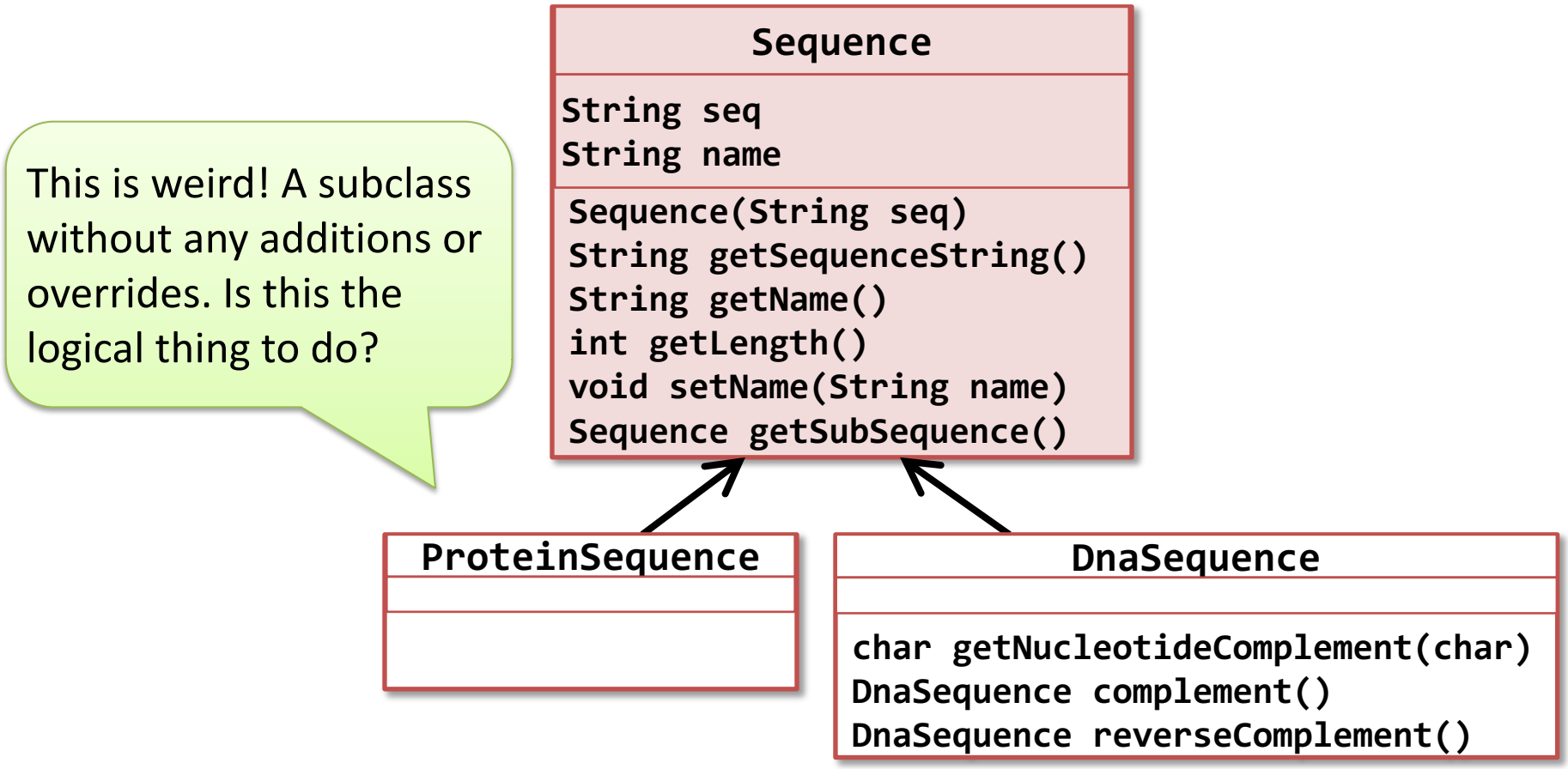
Extending the model

- OK, now we have a generic Sequence class in place
- But how about Proteins, and DNA?
- Do they fit seamlessly into this Sequence model?
- NO!
- Let's extend the model a bit to accommodate for the particulars of these sequence types

Extending the model



Extending the model



ProteinSequence

```
public class ProteinSequence extends Sequence{

   public ProteinSequence(String seqStr) {
     super(seqStr);
   }

   //put any protein-
   //specific code here
}
```

This calls the superclass constructor public Sequence(String seq)

Because you defined only a single constructor **Sequence(String seq){}** in class Sequence you have to implement the same in all subclasses, or call the super constructor with some default value (which isn't possible here).

DnaSequence

This is what a DnaSequence should be able to do in my view

```
public class DnaSequence extends Sequence {
  public DnaSequence(String seqStr) {
    super(seqStr);
  }
  /* returns a complement of a nucleotide character
   * and N if it is not a valid nucleotide */
  public static char getNucleotideComplement(char nuc) {
    // to be implemented later
    return 'N';
  }
  /* returns a complement copy of this
  public DnaSequence complement() {
    // to be implemented later
    return null;
  }
  /* returns a reverse complement copy
  public DnaSequence reverseComplement
    // to be implemented later
    throw new UnsupportedOperationException("not implemented yet");
  }
```

Again the mandatory constructor

While in development, I provide default return values …

Or better still, make it quite clear you're still in development

Complementing a nucleotide

```
/**
  * returns a complement c
  * and N if it is not
  */
 public static char getComplement(char nuc) {
   switch (nuc) {
     case 'A': return 'T';
     case 'a': return 'T';
     case 'C': return 'G';
     case 'c': return 'G';
     case 'G': return 'C';
     case 'g': return 'C';
     case 'T': return 'A';
     case 't': return 'A';
     default: return 'N';
   }
 }
```

Notice the **static** keyword? It says: I am class level, not object level. It makes this method available outside object context: **DnaSequence.getComplement('A');**

Here is a simple way to complement a nucleotide. We'll create a more efficient version later. When there is no valid DNA letter passed, I simply return N

More modifiers

- Let's revisit the old ones and introduce some new (and don't forget about ***default*** access)

keyword	meaning
private	not visible to anything but own object methods
public	visible to any method, any object, any class
protected	visible only to classes from the same package and to subclasses
static	methods and variables that are class-properties, not object properties
final	properties, methods, and classes with this keyword can not be changed, extended, overriden anymore

Static access

- The **`static`** keyword is very usefull if you want to make variables or methods accessible independent of objects
- when you make a method static you can call it on the class, as you have seen:

  `DnaSequence.getComplement('A');`
- There is one catch: static methods must be independent of object properties!

Final properties, methods and classes

- When you make a class final, it cannot be subclassed anymore
- When you make a method final, it cannot be overriden anymore
- When you make a variable final, it cannot be changed anymore. This is the same as a constant

Are you supposed to be created?

```
private void test(){
   Sequence seq = new Sequence("SOMERANDOMTEXT");
   seq.setName("Sequence1");
   System.out.println(seq.toString());
}
```

Don’t you think it is slightly funny that you can instantiate an object of type Sequence?

As with the generic Cell supertype before, Sequence should be made abstract

Complementing a DNA string

- Let's finally implement some DNA functionality

```
/* returns a complement copy of this sequ
public DnaSequence complement() {
  char[] arr = getSequenceString().toCharArray();
  char[] cArr = new char[arr.length];
  for (int i = 0; i < arr.length; i++) {
    cArr[i] = getComplement(arr[i]);
  }
  return new DnaSequence(new String(cArr));
}
```

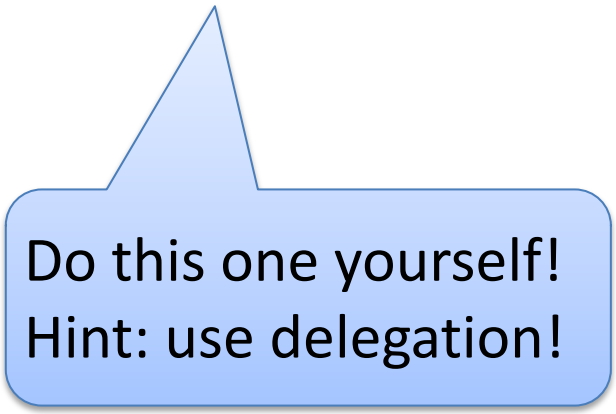
A nice method provided by class String

This is where the real complementing happens.

you can construct a String with a **char[]**

Reverse complementing a DNA string

```
/* returns a reverse complement sequence */
public DnaSequence reverseComplement() {
  //how would you do this?
}
```



DNA sequences

```
/* returns a reverse complement copy of this sequence */
public DnaSequence reverseComplement() {
  char[] arr = getSequenceString().toCharArray();
  char[] rcArr = new char[arr.length];
  for (int i = 0; i < arr.length; i++) {
    rcArr[i] = getComplement(arr[arr.length - (i+1)]);
  }
  return new DnaSequence(new String(rcArr));
}
```

This is where the reverse complementing happens. You can see this method is almost identical to `complement()`. ***Did you create a better implementation, without code duplication?***

Going for a test drive again

```
private void test() {
  DnaSequence dna = new DnaSequence("AGCCTATGGCATGCC");
  dna.setName("DNA1");
  DnaSequence dnaCompl = dna.complement();
  DnaSequence dnaRevCompl = dna.reverseComplement();
  System.out.println(dna.getSequenceString());
  System.out.println(dnaCompl.getSequenceString());
  System.out.println(dnaRevCompl.getSequenceString());
}
```

```
$ java SequenceTester
Sequence[name=Protein1 length=8]
AGCCTATGGCATGCC
TCGGATACCGTACGG
GGCATGCCATAGGCT
```

But how about RNA?

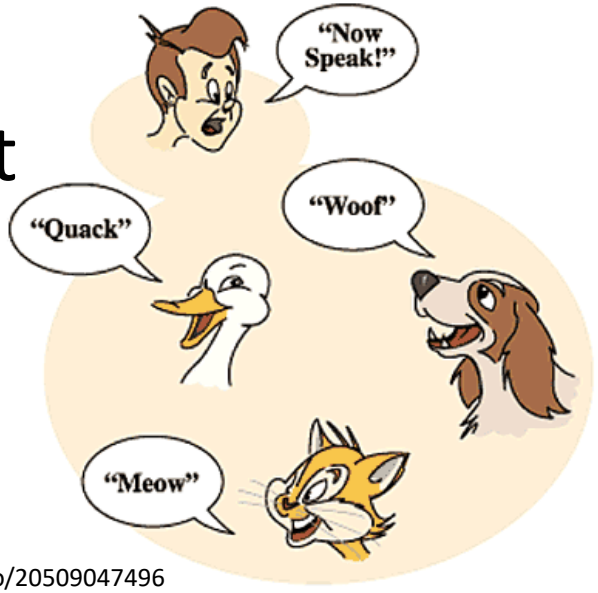
- You have an inheritance tree containing Sequence, Protein and DNA
- You probably know there is also RNA
- RNA and DNA are both nucleic acids. They have MUCH in common.

- Now modify your data model to also include RNA...
- And while you are at it, make some check, somewhere, on the legality of constructed sequence objects
- And we also forgot to test equals() and hashCode()…

PART 3: Polymorphism (the sequel)

Polymorphism revisited

- We are going to (re-)visit an aspect that makes Java (and OO in general) such a powerfull tool for creating complex applications: **polymorphism**
- Polymorphism means “being able to take many forms”
- This applied originally to biology; as you will see, this is no different



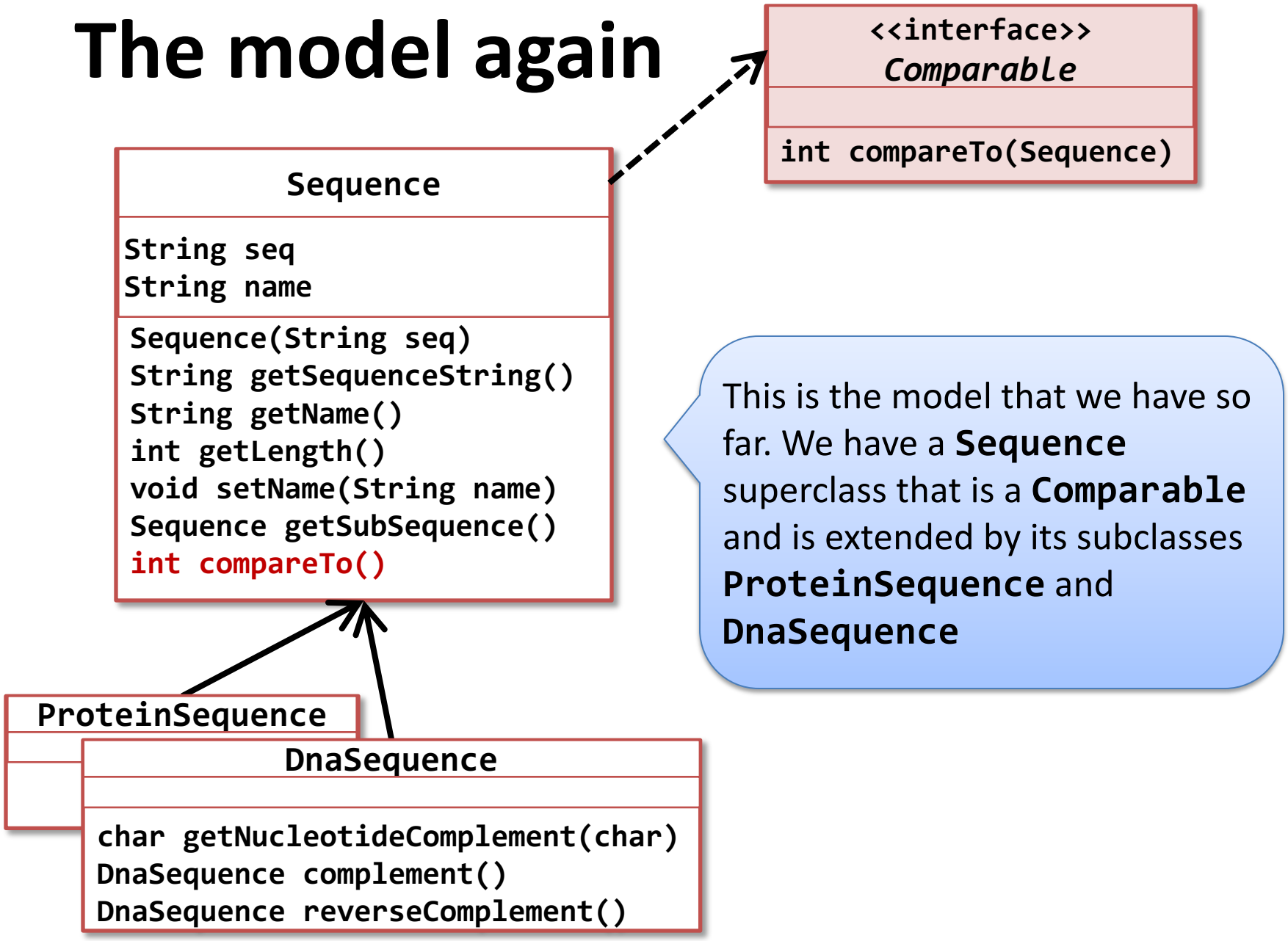
https://www.flickr.com/photos/globuloblanco/20509047496

Sorting is the new order

- In a previous session, we implemented some sorting functionality
- We'll use it in the next part of this presentation

```
/**implements Comparable.compareTo(), sorts by length,
 * shortest length will come first */
public int compareTo(Sequence otherSequence) {
    return this.getLength() - otherSequence.getLength();
}
```

The model again



The challenge: listing all types of sequences

- Suppose you have created a really funky graphical user interface for sequence analysis
- You want to be able to list all the sequences that the user has available, like this:

select	sequence name	type	length
☐	dnaPol1gene	DNA	2109
☑	dnaPol2gene	DNA	2242
☐	dnaPol1protein	Protein	761
☑	dnaPol2protein	Protein	788
☐	rnaPol1gene	DNA	1873

-

The bad solution

- You could do something like this:

```
ArrayList<ProteinSequence> prots =
                        dataAccessObject.getProteinSequences();
ArrayList<DnaSequence> dnas =
                        dataAccessObject.getDnaSequences();

graphicsController.listProteins(prots);
graphicsController.listDNAs(dnas);

/* etc etc, many many lines of code dealing identically
 * with very related objects. You already know what this
 * means:
 * MAINTENANCE NIGHTMARE
 */
```

The good solution

- There is a very nice way to deal with all your sequences in the same way: by dealing with them as Sequences!

```
ArrayList<Sequence> sequences =
                    dataAccessObject.getSequences();

graphicsController.listSequences(sequences);
```

You can treat ProteinSequences and DnaSequences as if they are Sequence objects. You can do this because a DnaSequence ISA Sequence and a ProteinSequence ISA Sequence

Meet polymorphism

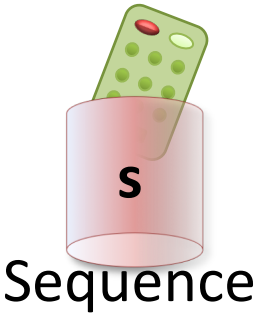
- This is the basis of polymorphism:

```
ArrayList<Sequence> sequences = new ArrayList<Sequence>();

Sequence dna1 = new DnaSequence("GATTACGGATAC");
sequences.add(dna1);

Sequence prot1 = new ProteinSequence("WGEASDTNY");
sequences.add(prot1);
```

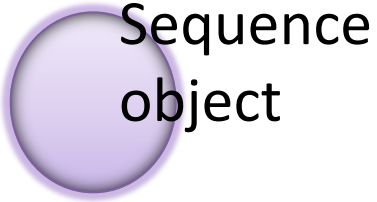
What happens here is you **declare a reference variable** of type Sequence, you **instantiate** an object of type **ProteinSequence** and make the Sequence variable **point to** the ProteinSequence object

The traditional way of creating object variables

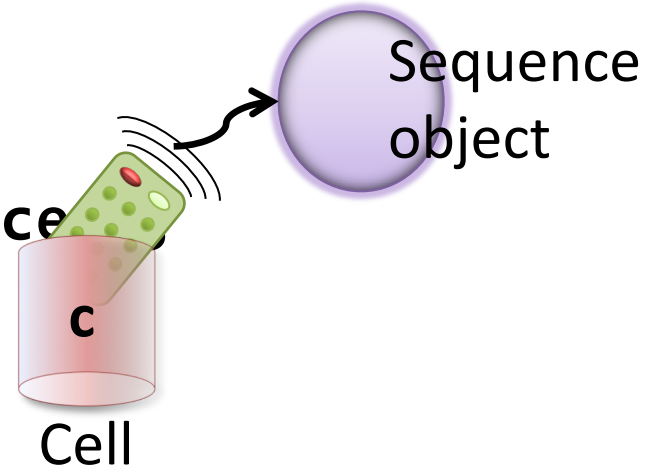
```
Sequence s = new Sequence();
```




```
Sequence c = new Sequence();
```

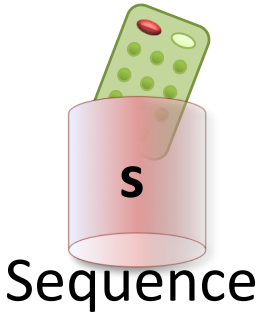



```
Sequence c = new Sequence();
```

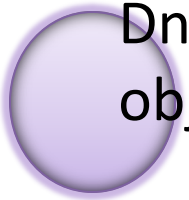


The polymorphic way of creating object variables

```
Sequence s
    = new DnaSequence();
```

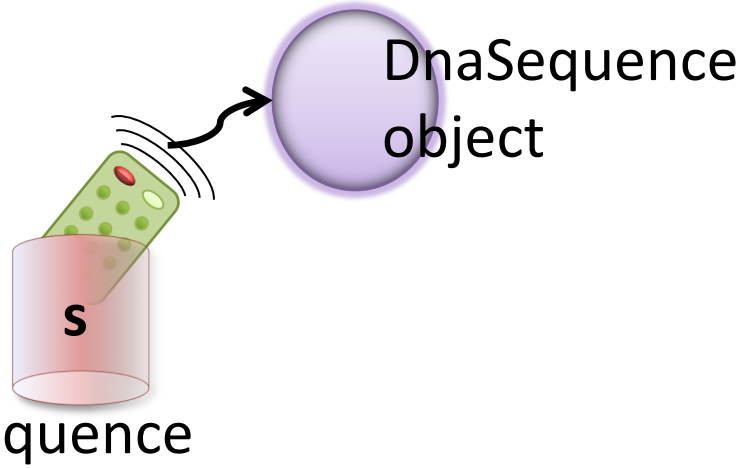



```
Sequence s
    = new DnaSequence();
```



Create the **polymorphic** object

```
Sequence s
    = new DnaSequence();
```



What does this make my sequence?

- Your Sequence reference variable still points to a real DnaSequence object; it has not been morphed or something
- **AN OBJECT WILL NEVER EVER CHANGE INTO ANOTHER OBJECT**

What is the consequence?

- You will only be able to call methods defined in the class of the reference variable
- You will be able to use the variable in collections of the supertype
- You will be able to use the variable in any context requesting the supertype (and you should define generic methods whenever possible)

What does this make my sequence?

```
Sequence sequence = new DnaSequence();
```

```
sequence.getSubSequence();
```

You **can** do this; it is a Sequence method

```
sequence.complement();
```

You **can't** do this; it is a DnaSequence method

How do I get my DnaSequence back?

- If you want your DnaSequence functionality back, you'll have to cast it to the desired type:

```
Sequence sequence = new DnaSequence();
sequence.getSubSequence();
/*you want to complement it*/
DnaSequence complement =
        (DnaSequence) sequence.complement();
```

DNA will never become protein

- There is one big risk involved: you must be **sure** you are casting an object to its correct type, otherwise you get the dreaded *ClassCastException*
- To be sure of the correct cast, you can put in this test:

`instanceof` is a binary operator that returns true if an object is of the given type

```
If (sequence instanceof DnaSequence) {
   DnaSequence =
      (DnaSequence) sequence.complement();
}
```

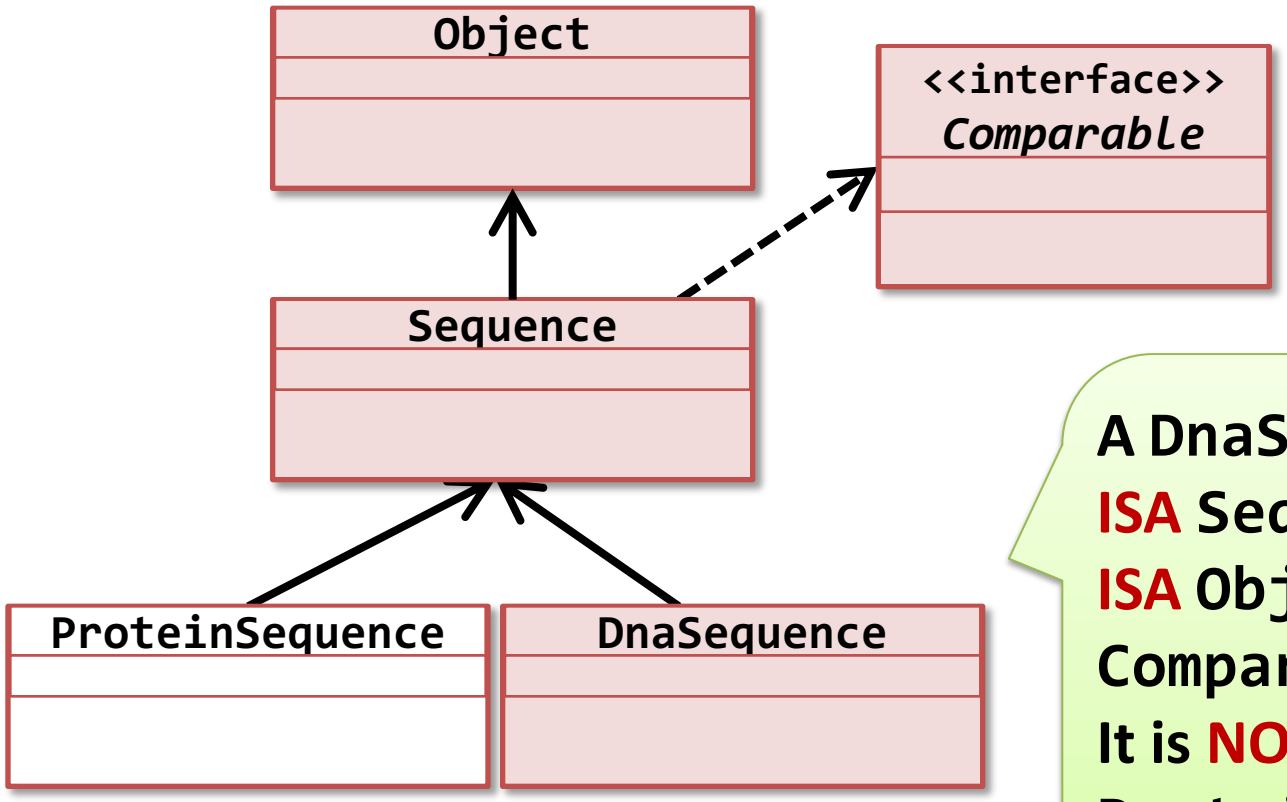
Instanceof demo

```
import static java.lang.System.out; //for brevity only!!
Sequence seq = new DnaSequence("GATTGCTCGGGATA") ;
seq.setName("dna1");

out.println(seq.toString());
out.println("seq instanceof Sequence: " + (seq instanceof Sequence));
out.println("seq instanceof DnaSequence: " + (seq instanceof DnaSequence));
out.println("seq instanceof ProteinSequence: " + (seq instanceof
                                                  ProteinSequence));
out.println("seq instanceof Object: " + (seq instanceof Object));
out.println("seq instanceof Comparable: " + (seq instanceof Comparable));
```

```
$ java PolymorphismDemo
Sequence[name=dna1 length=14]
seq instanceof Sequence: true
seq instanceof DnaSequence: true
seq instanceof ProteinSequence: false
seq instanceof Object: true
seq instanceof Comparable: true
```

Yes, you can even treat a DnaSequence object as if it is only a Comparable !!

ISA



A DnaSequence ISA Sequence, ISA Object and ISA Comparable.
It is NOT A ProteinSequence !!

The power of polymorphism

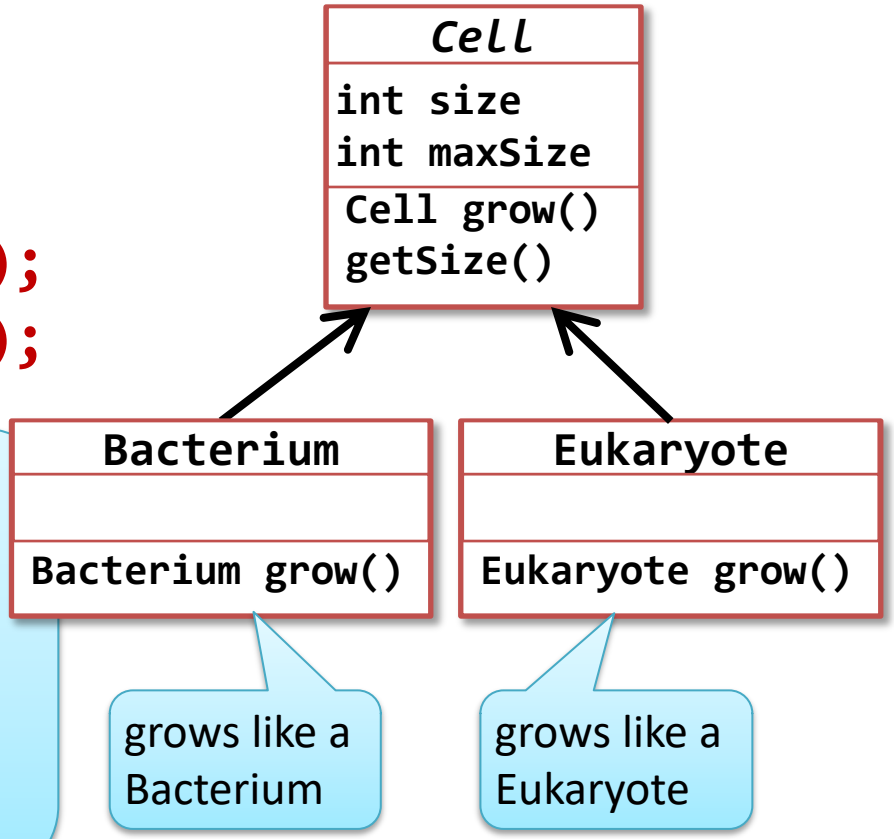
Using inheritance and polymorphism, you can

- simply deal with different types of objects identically **(polymorphic reference)**
- extend the model easily with other classes without having to change/add code all over your application
- implement different behaviours behind a method defined in a superclass: **polymorphic behaviour**

Remember your testtube and cells?

```
Cell cell = new Bacterium();
Cell babyCell = cell.grow();
```

grow() has been declared in Cell. It has been **overriden** in Bacterium and Eukaryote to reflect their different growing styles. This is also an aspect of polymorphism: **polymorphic behaviour**

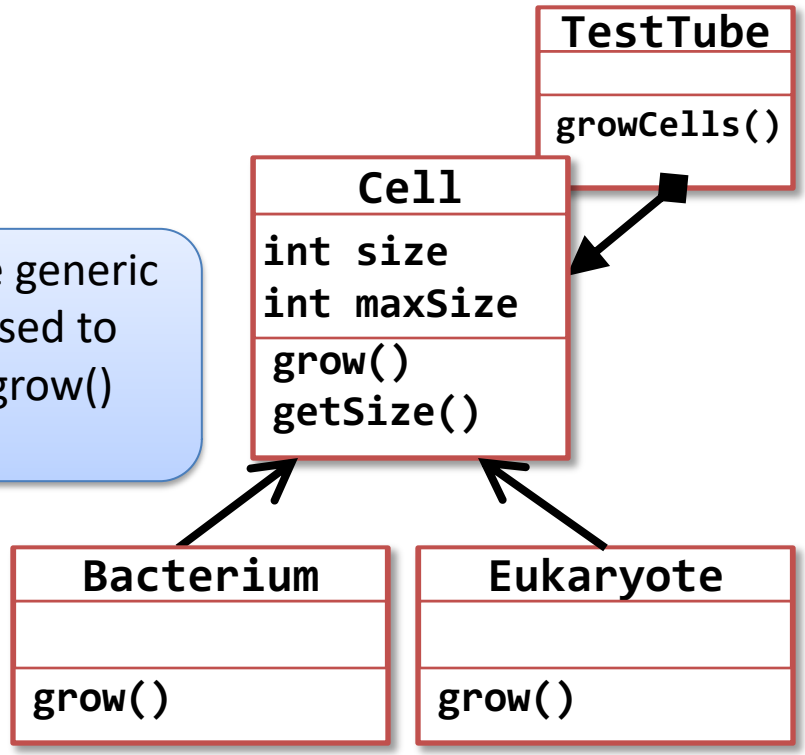


a polymorphic testtube

```
public class TestTube {
    public static void main(String[] args) {
        TestTube testTube = new TestTube();
        testTube.growCells("bacterium");
    }

    public void growCells(String type) {
        Cell cell;
        if (type.equals("bacterium")) {
            cell = new Bacterium();
        }
        else if (type.equals("eukar
            cell = new Eukaryote();
        }
        //room for other types such
        //as Archaea, or bone cells
        Cell babyCell = cell.grow();
    }
```

The generic type Cell is declared, but specific types are instantiated (Bacterium, Eukaryote)

Here the generic type is used to call the grow() method



polymorphic behaviour IN CODE

```
public abstract class Cell {
    private int size;
    private int maxSize;
    /*returns the cell's size*/
    public int getSize(){
        return size;
    }
    /*abstract method!*/
    public abstract Cell grow();
  }
}
```

```
public class Bacterium extends Cell {
    /*overrides Cell.grow()*/
    public Cell grow() {
        System.out.println("a bacterium growing");
    }
}
```

```
public class TestTube{
    public static main( String[] args){
        Cell cell;
        //cell = new Cell(); ILLEGAL
        //cell.grow();
        cell = new Bacterium();
        cell.grow();
    }
}
```

Calls grow() on a Bacterium object.

```
$ java TestTube
a bacterium growing
```

PART 3: Loose ends

Some loose ends

- There are actually some subjects that were skipped so far
- Since they are so important for well-designed applications, they will revisited here shortly:
  - Interfaces
  - Enums
  - Inner classes

Interfaces

- You of course remember the Comparable interface: It is a contract that you can sign within your classes
- Signing it will force you to live up to the contract by implementing the compareTo() method
- When you sign the contract, your class can be (polymorphically) treated as an instance of the interface

Interfaces

- Interfaces are like all-abstract classes, but with one very important distinction:
- Classes can extend only a single class, but as many interfaces as they like!
- In Object-oriented programming there is a very important design rule that says ***"Program against interfaces, not against implementations"***
- Remember this rule, and you'll always be a happy camper when major refactoring is required

Enums revisited

- Quite often, you have methods that make decisions based on some variable that has only a few allowed values
- Inexperienced programmers often use **`public static final`** constants for these well-defined values
- When you are truly a Java programmer, you will use the Force of ***enums***

Enums: constants on steroids

```
public enum UserType{
    GUEST,
    USER,
    HACKER;
}
public void greetUser(UserType userType){
    switch(userType){
    case GUEST:
        System.out.println("Register so we can spam you");
        break;
    case USER: System.out.println("Thanks, spam is underway");
        break;
    case HACKER: System.out.println("I cannot spam you");
    }
}
```

Only three types of Users can ever exist

Easy switching on all types!

Enums: constants on steroids

```
public enum LifeDomains {
     BACTERIA("bacteria"),
     ARCHAEA("archaea"),
     EUKARYA("eukarya"){
          public boolean isProkaryote() { return false; }
     };
     private String type;
     private LifeDomains(String type) {
          this.type = type;
     }
     public String toString() { return type; }
     public boolean isProkaryote() { return true; }
}
public static void main(String[] args) {
     System.out.println(LifeDomains.EUKARYA);
     System.out.println(LifeDomains.EUKARYA.isProkaryote());
}
```

A within-class override of the default implementation: a **constant-specific class body**!

A regular method

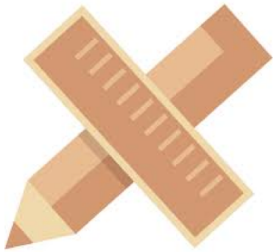
Inner classes

- While we have done so almost always, there is a widespread misconception about how many classes can exist in one file
- There can be only one ***public*** top-level class per file
- A file can, however, have many (anonymous) inner classes and interface implementers:

Inner classes

```
public class Outer {
        private Usable usable = new Usable() {
                public void use() { }
        };
        public void useInners() {
                Inner i = new Inner();
                usable.use();
        }
        private class Inner {
        /* extra special class dedicated to
         * functioning of Outer class*/
        }
        private interface Usable {
                public void use();
        }
}
```

Take a minute to study this and realize the myriad of design choices it gives you



Design rule

- You make a class an inner classes when
  - this class is very tightly bound to the outer class (logically, functionally)
  - this class has no reason to live outside the outer class
- Make a clear distinction between static and non-static inner classes!

that's it folks!

- You have reached the end of the course and are close to becoming a kick-butt Java programmer
- remember to drop the buzz words with your supervisor now and then:
  - **inheritance**
  - **encapsulation**
  - **polymorphism**
  - **and the best of all: Constant-specific class body**

Would you hire you?

- Test yourself at
- http://www.toptal.com/java#hiring-guide