CS 5004: Lecture 8

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- Collaborating with Github
- Parametric polymorphism (AKA generics)

Collaborating with GitHub

Now you're working in groups...

Good git hygiene is vital!

Why? Merge conflicts ☹

Avoiding merge conflicts

Golden rule #1: Do not touch code/files that someone else is likely to edit

If a teammate is working on SomeClass.java, DO NOT

- edit SomeClass.java
- move SomeClass.java
- rename/change the package containing SomeClass.java

Discuss who is working on what <u>before</u> touching anything!

Avoiding merge conflicts

Golden rule #2: ALWAYS start work with the following commands

```
git status
<commit and push any outstanding local changes>
git pull
```

Even if you only left your computer for a short time and did not shut your computer/files e.g.

- Left the IDE open overnight
- Went to the store
- Made a sandwich

Avoiding merge conflicts

Golden rule #3: NEVER work directly on the master branch

When beginning a new assignment, create a new “topic branch” just for you (instructions on Canvas)

- Do all your work on that branch
- Merge to master only when your topic branch is working without compile time errors and is thoroughly tested

If a merge conflict happens...

1) Don’t panic

2) DO NOT use –f or --force

No matter how many Stack Overflow posts tell you to do it!

Practice with branches

In your own repo, using the command line (not the GUI or IntelliJ):

`cd path/to/your/repo`

`git status` (commit and push any outstanding changes, if needed)

`git pull`

`git checkout –b branch_name`

Write some code, commit and push

→ you will need to set the remote branch on first push:

`git push --set-upstream origin branch_name`

Parametric polymorphism

AKA Generics

This will not be on the exam!

Required for HW7

What exactly is being polymorphic

So far:

- Objects
  - Instance of subclass (e.g. Cat) treated as instance of super class (e.g. Animal)
- Methods/constructors – overloading

What exactly is being polymorphic

Parametric polymorphism (generics):

- "Enables **types** (classes and interfaces) to be **parameters** when defining classes and interfaces."
- Especially useful when writing classes that are collections of other objects (e.g. List, Set, Stack etc).
  - Write one class that can handle multiple types of objects.

Enables a function or class to be written such that it handles values identically regardless of type

Example: List ADT

An ordered collection (also known as a sequence)

- A user controls where in the list each element is inserted
- A user can access elements by their integer index (position in the list)

Example: List ADT

An ordered collection (also known as a sequence)

- Doesn't specify what **type of data** can be stored in the List

Example: ILinkedList

Shares some functionality with the List ADT but can **only** contain Integers

- not Strings
- not Cats etc

Java’s built-in collections

…don’t specify the type of data to be stored in the collection

Class ArrayList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Direct Known Subclasses:

`AttributeList, RoleList, RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Java's built-in collections

`<E>` - shows this is a **generic** data structure (you can put almost anything in it) e.g.

- Replace the E with the type you want to store

```
ArrayList<String> strList
      = new ArrayList<String>();
```

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Type parameters

```
List<Type> name = new ArrayList<Type>();
```

Type parameter specifies type of element stored in the collection

- Allows the same class to store different types of objects
- Also called a *generic* class

```
List<String> names = new ArrayList<String>();
List<Integer> digits = new ArrayList<Integer>();
```

What can be a type parameter?

Objects only

- Setting a primitive as a type parameter → compile time error e.g.
```
List<int> digits = new ArrayList<int>(); //won't compile
```
- Instead, use a wrapper class type:

Primitive	Wrapper
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Using type parameters: a shortcut

Right side Type argument is unnecessary:

```
List<Type> name = new ArrayList<Type>();
```

Instead, use the diamond operator, **<>**:

```
List<Type> name = new ArrayList<>();
```

Compiler auto populates each type parameter from the types on the left side

```
List<String> names = new ArrayList<>();
```

Generic Linked List ADT – a list of (almost) anything

Support the following operations:

- `count` – get the number of items in the list
- `getItem` – get the item in the current node
- `getRest` – get the rest of the list
- `insert` – insert an item at the head of the list
- `insertAt` – insert an item at a specific index

Generic Linked List ADT – the interface

```
public interface ILinkedList<T> {
  Integer count();
  T getItem();
  ILinkedList getRest();
  ILinkedList insert(T item) throws IndexOutOfBoundsException;
  ILinkedList insertAt(T item, Integer index)
                                  throws IndexOutOfBoundsException;
}
```

Generic List ADT – the interface

```
public interface ILinkedList<T> {
  Integer count();
  T getItem();
  ILinkedList getRest();
  ILinkedList insert(T item) throws IndexOutOfBoundsException;
  ILinkedList insertAt(T item, Integer index)
                                  throws IndexOutOfBoundsException;
}
```

A placeholder for the datatype that will be stored in the list

Generic List ADT – the interface

```
public interface ILinkedList<T> {
  Integer count();
  T getItem();
  ILinkedList getRest();
  ILinkedList insert(T item) throws IndexOutOfBoundsException;
  ILinkedList insertAt(T item, Integer index)
                                  throws IndexOutOfBoundsException;
}
```

Use the placeholder anywhere you need to indicate type

`<placeholder>` naming conventions

You *could* call it whatever you want e.g. `<insertNameHere>`

- Convention - a single uppercase letter

Common parameter names:

- E – Element
- T – Type
- K – Key

...and more

Implementing the ILinkedList

IntelliJ will auto-generate methods with “T” replaced with “Object”…

```
ILinkedList insert(Object item) {
   …
}
Object getItem() {
   …
}
```

Implementing the ILinkedList

IntelliJ will auto-generate methods with “T” replaced with “Object”…

```
ILinkedList insert(Object item) {
   …
}
Object getItem() {
   …
}
```

- **A problem for clients**
- Will not enforce type requirements → runtime errors that are hard to detect.

Converting from Object to generic <T>

```
public class EmptyNode implements ILinkedList {
  Integer count() {...}
  Object getItem() {...}
  ILinkedList getRest() {...}
  ILinkedList insert(Object item) {...}
  ILinkedList insertAt(Object item, Integer index) {...}
}
```

Converting from Object to generic <T>

```
public class EmptyNode<T> implements ILinkedList<T> {
  Integer count() {...}
  Object getItem() {...}
  ILinkedList getRest() {...}
  ILinkedList insert(Object item) {...}
  ILinkedList insertAt(Object item, Integer index) {...}
}
```

Change the header to indicate this class takes generic parameters

- Note the triangle brackets <>
- Object data types only (i.e. not primitives)

Converting from Object to generic <T>

```
public class EmptyNode<T> implements ILinkedList<T> {
  Integer count() {...}
  Object getItem() {...}
  ILinkedList<T> getRest() {...}
  ILinkedList<T> insert(Object item) {...}
  ILinkedList<T> insertAt(Object item, Integer index) {...}
}
```

Add <T> to all references to ILinkedList

Converting from Object to generic <T>

```
public class EmptyNode<T> implements ILinkedList<T> {
  Integer count() {...}
  T getItem() {...}
  ILinkedList<T> getRest() {...}
  ILinkedList<T> insert(T item) {...}
  ILinkedList<T> insertAt(T item, Integer index) {...}
}
```

Change all “Objects” to the generic type parameter, “T”

Using a generic type

Specify **`T`** when declaring and instantiating:

```
ILinkedList<Integer> intList = new EmptyNode<>();
ILinkedList<Cat> catList = new EmptyNode<>();
```

Using a generic type

Specify **`T`** when declaring and instantiating:

```
ILinkedList<Integer> intList = new EmptyNode<>();
ILinkedList<Cat> catList = new EmptyNode<>();
```

…will enforce type requirements in any methods that have **`T`** as a parameter

Guaranteeing type safety

```
ILinkedList<Book> bookList = new EmptyNode<Book>();
Book book = new Book("A Book", "An Author", 2019, 8.99f);
bookList = bookList.append(book);
bookList = bookList.append("Not a Book");
```

→ **compile-time error**

Exercise 2: Make Stack and Queue generic

- Add generic parameters to Stack and Queue
  - Existing Stack / Queue may show errors before parameters are added
- Modify tests as needed
- You might have “unchecked” Xlint warnings when tests are run
  - Can you figure out why?

Break (10 mins)

Generic class from scratch: Vet clinic example

Software to manage a vet's patient list

Each vet has:

- a maximum number of patients
- a specialty e.g.
  - domestic animals
  - farm animals
  - birds

Vet clinic example: animals

Vet clinic example: using inheritance

In `AbstractVet`:

- Patients stored in `List<AbstractNonHuman>`
  - Ensures only animals added to the list
- Specialty encoded as an enum

Vet clinic example: using inheritance

Adding a patient → must ensure the patient matches the specialty

```
public boolean canAcceptPatient(AbstractNonHuman animal) {
    // Not extensible! What if new species categories are added?
    if (this.specialty == Specialty.DOMESTIC)
        return (animal instanceof AbstractDomestic);
    else if (this.specialty == Specialty.FARM)
        return (animal instanceof AbstractFarm);
    else if (this.specialty == Specialty.BIRD)
        return (animal instanceof AbstractBird);
    return false;
}
```

A generic PatientList

Create a new generic class to:

- **encapsulate** the maximum number of patients a vet can have *and* their patient information
- **restrict** patients to the appropriate species/category

```
PatientList<Cat> catsOnly = new PatientList<>(100);
PatientList<AbstractFarm> farmPatients = new PatientList<>(20);
```

A generic PatientList

```
public class PatientList<T> {
  private int maxPatients;
  private List<T> patients;
  public PatientList(int maxPatients) {
    this.maxPatients = maxPatients;
    this.patients = new ArrayList<>();
  }
  public List<T> getPatients() {
    return this.patients;
  }
  public void addPatient(T patient) {
    this.patients.add(patient);
  }
}
```

A generic PatientList

```
public class PatientList<T> {
  private int maxPatients;
  private List<T> patients;
  public PatientList(int maxPatients) {
    this.maxPatients = maxPatients;
    this.patients = new ArrayList<>();
  }
  public List<T> getPatients() {
    return this.patients;
  }
  public void addPatient(T patient) {
    this.patients.add(patient);
  }
}
```

A placeholder for the datatype that will be stored in the list

A generic PatientList

```
public class PatientList<T> {
  private int maxPatients;
  private List<T> patients;
  public PatientList(int maxPatients) {
    this.maxPatients = maxPatients;
    this.patients = new ArrayList<>();
  }
  public List<T> getPatients() {
    return this.patients;
  }
  public void addPatient(T patient) {
    this.patients.add(patient);
  }
}
```

Use the placeholder anywhere you need to indicate generic type

Multiple generic parameters

Design a class that can hold **any pair of objects**

For example:

- First name and last name
- Birth month (Jan... Dec) and birth day (1...31)
- X and Y coordinates

Multiple generic parameters

```
public class Pair<T, U> {
  private T first;
  private U second;

  public Pair(T first, U second) { ... }

  public T getFirst() { ... }

  public U getSecond() { ... }
}
```

List multiple params

- Must have different names, even if types might be the same

Extending a generic class

```
public class Point2D extends Pair<Double, Double> {
  public Point2D(Double x, Double y) {
    super(x, y);
  }

  public Double getX() { return super.getFirst() }

  public Double getY() { return super.getSecond() }
}
```

Extending a generic class

```
public class Point2D extends Pair<Double, Double> {
  public Point2D(Double x, Double y) {
    super(x, y);
  }

  public Double getX() { return super.getFirst() }

  public Double getY() { return super.getSecond() }
}
```

Generic placeholders replace with actual types

Extending a generic class

```
public class Point2D extends Pair<Double, Double> {
  public Point2D(Double x, Double y) {
    super(x, y);
  }

  public Double getX() { return super.getFirst() }

  public Double getY() { return super.getSecond() }
}
```

More meaningful getter names...

Extending a generic class

```
public class Point2D extends Pair<Double, Double> {
  public Point2D(Double x, Double y) {
    super(x, y);
  }

  public Double getX() { return super.getFirst() }

  public Double getY() { return super.getSecond() }
}
```

More meaningful getter names...
.... still call the inherited methods

Exercise 3

Write a HighScore class that extends the Pair class

HighScore contains:

- username: a String
- score: an Integer
- Getters for each of the above

Setting boundaries

```
public class PatientList<T> { ... }
```

Compiler will allow PatientList to contain **any object**

```
PatientList<Cat> catsOnly = new PatientList<>(100);
PatientList<Book> books = new PatientList<>(100);
```

Setting boundaries

If type is not specified → defaults to **`T (Object)`** e.g.

```
Person doolittle = new Person("Dr.", "Doolittle");
Cat mittens = new Cat("Mittens", doolittle);

PatientList patients = new PatientList(10);
patients.addPatient(doolittle);
patients.addPatient(mittens);
```

Setting boundaries

If type is not specified → defaults to **`T (Object)`** e.g.

```
Person doolittle = new Person("Dr.", "Doolittle");
Cat mittens = new Cat("Mittens", doolittle);

PatientList patients = new PatientList(10);
patients.addPatient(doolittle);
patients.addPatient(mittens);
```

Treated as class Object

- Actual class methods no longer available
- Type erasure

Bounded type parameters

Restrict the types that can be passed to a class by **bounding** the type parameter:

```
<T extends ClassName>
```

Bounded type parameters

Restrict the types that can be passed to a class by **bounding** the type parameter:

```
<T extends ClassName>
```

Only objects that are type **`ClassName`** can be passed to the class.

- Always **`extends`**, even if **`ClassName`** is an interface

Bounding the PatientList class

```
public class PatientList<T extends AbstractNonHuman> {
  private int maxPatients;
  private List<T> patients;
  public PatientList(int maxPatients) {
    this.maxPatients = maxPatients;
    this.patients = new ArrayList<>();
  }
  public List<T> getPatients() {
    return this.patients;
  }
  public void addPatient(T patient) {
    this.patients.add(patient);
  }
}
```

Only need **`extends`**… in the the header

- Anywhere there's a **`T`** will have compile-time type of **`AbstractNonHuman`**

Bounding the PatientList class

If type is not specified → defaults to **`AbstractNonHuman`** e.g.

```
Person doolittle = new Person("Dr.", "Doolittle");
Cat mittens = new Cat("Mittens", doolittle);

PatientList patients = new PatientList(10);
patients.addPatient(doolittle);
patients.addPatient(mittens);
```

Compile time error!

- A Person is not an AbstractNonHuman

A limitation for the vet case

In real life, vets may be qualified to treat multiple types of animal that don't correspond to the inheritance tree

E.g. birds & domestics but not farm animals

No way to represent this using generics alone!

When are generics most useful?

Generics are most useful for:

- Collections of things – standard functionality, common to all types
- Generic algorithms e.g. sorting → generic methods

Generic methods

- Allow you to write one method that can handle different argument types
- Can (sometimes) be used instead of method overloading
  - Most useful for methods that act on arrays/collections

Generic methods example

Imagine we want to print all items of an array in a particular format

- Could overload a method – one version per array type
- …redundant code

```
public void printArr(Integer[] arr) {
  for (int i = 0; i < arr.length; i++)
{
    System.out.println(i + ": "
                              + arr[i]);
  }
}
public void printArr(String[] arr) {
  for (int i = 0; i < arr.length; i++)
{
    System.out.println(i + ": "
                              + arr[i]);
  }
}
```

Generic methods example

Or we could use generics and write one method for all arrays…

```
public <E> void printArr(E[] arr) {
  for (int i = 0; i < arr.length; i++) {
    System.out.println(i + ": " + arr[i].toString());
  }
}
```

Generic methods example

```
public <E> void printArr(E[] arr) {
  for (int i = 0; i < arr.length; i++) {
    System.out.println(i + ": " + arr[i].toString());
  }
}
```

Indicate this is a generic method in the method header

- Goes before the return type
- (It is not the return type!)

Generic methods example

```
public <E> void printArr(E[] arr) {
  for (int i = 0; i < arr.length; i++) {
    System.out.println(i + ": " + arr[i].toString());
  }
}
```

Use the type placeholder in the parameters

Generic methods – returning a generic

```
public <E> E lastItem(E[] arr) {
  int lastIndex = arr.length – 1;
  return arr[lastIndex];
}
```

What is this method doing?
…and what is it returning?

Calling generic methods

```
MyClass myVar = new MyClass();
String[] strings = {"A", "B", "C"};
myVar.printArr(strings);
// prints:
0: A
1: B
2: C
```

Called in the same way as any other method:

- Instantiate a new object of the class

Calling generic methods

```
MyClass myVar = new MyClass();
String[] strings = {"A", "B", "C"};
myVar.printArr(strings);
// prints:
0: A
1: B
2: C
```

Called in the same way as any other method:

- Instantiate a new object of the class
- Call the method using **`objectName.methodName(params);`**

Calling generic methods

```
MyClass myVar = new MyClass();
String[] strings = {"A", "B", "C"};
myVar.printArr(strings);
// prints:
0: A
1: B
2: C
```

Called in the same way as any other method:

- Instantiate a new object of the class
- Call the method using **`objectName.methodName(params);`**
- Compiler checks that params meet the placeholder needs:
  - Inherit Object if unbounded
  - Inherit the given class if bounded

Static methods with generics

Sometimes it doesn't makes sense to instantiate a new object just to call a method.

- e.g. if the method doesn't reference a property belonging to the class.

```
public <E> void printArr(E[] arr) {
  for (int i = 0; i < arr.length; i++) {
    System.out.println(i + ": " +
                arr[i].toString());
  }
}
```

Static methods with generics

Make these methods static so they can be used without creating an unnecessary Object.

- Static methods must be “standalone”--can’t access non-static properties or methods

```
public static <E> void printArr(E[] arr)
{
  for (int i = 0; i < arr.length; i++) {
    System.out.println(i + ": " +
                arr[i].toString());
  }
}
```

Static methods with generics

Call a static method without creating an instance of the class:

- `ClassName.methodName(params);`
- `ClassName.printArr(anArray);`

```
String[] strings = {“A”, “B”, “C”};

ArrayHelper myVar = new ArrayHelper();
myVar.printArr(strings);
```

…becomes…

```
ArrayHelper.printArr(strings);
```

Wildcards

? is used in generic code to represent an **unknown** type

- Used in methods (return or parameter type), not class headers

Wildcard example

`equals()` in **`PatientList`**

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  PatientList<?> that = (PatientList<?>) o;
  return maxPatients == that.maxPatients &&
           currentPatients.equals(that.currentPatients);
}
```

Another wildcard example

`foo` accepts an ArrayList containing objects of unknown type

```
public void foo(ArrayList<?> things) {
  for (   thing : things) {

    System.out.println(thing.toString()
                        + " is a thing");

  }
}
```

Another wildcard example

`foo` accepts an ArrayList containing objects of unknown type

- Indicates the wildcard in the parameter.

```
public void foo(ArrayList<?> things) {
  for (   thing : things) {

    System.out.println(thing.toString()
                        + " is a thing");

  }
}
```

Another wildcard example – client method

Still need to indicate type here so Java knows how to treat **`thing`**

```
public void foo(ArrayList<?> things) {
  for (   thing : things) {
    System.out.println(thing.toString()
                       + " is a thing");
  }
}
```

Another wildcard example – client method

Still need to indicate type here so Java knows how to treat **`thing`**

- Can't use **`?`**, it's a placeholder
- Will be the base type – **`Object`**

```
public void foo(ArrayList<?> things) {
  for (Object thing : things) {
    System.out.println(thing.toString()
                       + " is a thing");
  }
}
```

Another wildcard example – client method

Still need to indicate type here so Java knows how to treat **`thing`**

- Can't use **`?`**, it's a placeholder
- Will be the base type – **`Object`**
  - An **unbounded** wildcard

```
public void foo(ArrayList<?> things) {
  for (Object thing : things) {
    System.out.println(thing.toString()
                       + " is a thing");
  }
}
```

Bounded wildcards

Wildcards can be bounded just like class parameters:

- ? is an unknown type of at least type **`Animal`** (i.e. it is **`Animal`** or it inherits **`Animal`**).
- An **upper bounded** wildcard

```
public void foo(
  ArrayList<? extends Animal> things) {
  for (Object thing : things) {

    System.out.println(thing.toString()
                       + " is a thing");

  }
}
```

Bounded wildcards

Wildcards can be bounded just like class parameters:

- ? is an unknown type of at least type **`Animal`** (i.e. it is **`Animal`** or it inherits **`Animal`**).
- An **upper bounded** wildcard

```
public void foo(
  ArrayList<? extends Animal>
things) {
  for (Animal thing : things) {

System.out.println(thing.toString()
                        + " is a
thing");
  }
}
```

Change to upper bound type, Animal.

- Could be anything lower down the inheritance tree (e.g. Cat)
- … but not anything higher up (e.g. Object)

Bounded wildcards

`super` instead of **`extends`**:

- ? is an unknown type of **`Cat`** or above (i.e. Cat, AbstractAnimal, Object…excludes sibling, Dog).
- A **lower bounded** wildcard

```
public void foo(
  ArrayList<? super Cat> things) {
  for (Object thing : things) {

    System.out.println(thing.toString()
                       + " is a thing");

  }
}
```

Bounded wildcards

`super` instead of **`extends`**:

- ? is an unknown type of **`Cat`** or above (i.e. Cat, AbstractAnimal, Object…excludes sibling, Dog).
- A **lower bounded** wildcard

```
public void foo(
  ArrayList<? super Cat> things) {
  for (Object thing : things) {

System.out.println(thing.toString()
                        + " is a
thing");
  }
}
```

In this case, `thing`'s type must be `Object`

- Could be anything higher up the inheritance tree (e.g. Object)
- … but not anything more specific

Type erasure

= how Java compiles generic placeholders and wildcards

- All placeholders and wildcards are replaced with either Object (if unbounded) or the bound class (if bounded)
- **`<T>`** compiles as **`Object`**
- **`<T extends AbstractAnimal>`** compiles as **`AbstractAnimal`**

Type erasure & overloading

Can’t use method overloading with generic parameters if multiple signatures compile as the same type e.g.:

```
public void print(List<String> list) {...};
public void print(List<Integer> list) {...};
```

Type erasure & overloading

Can't use method overloading with generic parameters if multiple signatures compile as the same type e.g.:

```
public void print(List<String> list);
public void print(List<Integer> list);
```

If the generic parameter is unbounded <T> → both compile to Object

Caution: generics and arrays

You cannot create objects or arrays from a parameterized type

```
public class Foo<E> {
  private E myField;
  public void method1(E param) {
    myField = new E(); // Error!
    E[] a = new E[10]; // Error!
  }
}
```

Caution: generics and arrays

You <u>can</u> accept values of that type

```
public class Foo<E> {
  private E myField;
  public void method1(E param) {
    myField = param;
    E[] a = new E[10]; // Error!
  }
}
```

Caution: generics and arrays

You can create arrays by casting from Object

```
public class Foo<E> {
  private E myField;
  public void method1(E param) {
    myField = param;
    E[] a = (E[]) (new Object[10]);
  }
}
```

...but this approach has disadvantages (unchecked casting)

- Avoid using generic arrays!

Caution: generics and arrays

For example:

```
static <E> E[] createArray(int size) {
  return (E[]) new Object[size];
}

public static void main(String[] args) {
  String[] array = createArray(10); // Throws ClassCastException
}
```

Caution: Generics and polymorphism

Generic collections are NOT polymorphic on the type.

For example:

```
List<Object> words = new ArrayList<String>() // ERROR
List<AbstractAnimal> animals
                          = new ArrayList<Cat>() // ERROR
```

Caution: Generics and polymorphism

Generic collections are NOT polymorphic on the type.

For example:

```
List<Object> words = new ArrayList<String>() // ERROR
List<AbstractAnimal> animals
                          = new ArrayList<Cat>() // ERROR
```

These must match!

Extension: doctors and vets

Now the company wants to support (human) doctors too!

Vets & doctors both have:

- a max number of patients
- a patient list

Key difference:

- Vets can only treat non humans
- Doctors can only treat humans

Exercise 4: doctors and vets

Refactor PatientList (genericpatientslist) to use generics to work for doctors and patients alike → Use generics!

Key difference:

- Vets can only treat non humans
- Doctors can only treat humans