CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2024

LECTURE 8

Tamara Bonaci
t.bonaci@northeastern.edu

Northeastern University
Khoury College of Computer Sciences

AGENDA

- Review: Well-designed OOD systems
- Collaborating with GitHub
- Parametric polymorphism – generics
  - Generic linked list
  - Example – vet clinic
  - Generics – part 2
  - Bounded data parameters
  - Generic methods
  - Wild cards
  - Type erasure
  - Generics and subtyping

COURSE LOGISTICS

REVIEW: WELL-DESIGNED (OOD) SYSTEMS

- Object Oriented Design Principles
- SOLID principles

REVIEW: WELL-DESIGNED SYSTEM: SOLID PRINCIPLES

- S – Single responsibility principle
  - (one class, one responsibility)
- O – open closed principle
  - (open for extension, closed for modification)
- L – Liskov substitution principle
  - (derived classes must be substitutable for their base classes)
- I – Interface segregation principle
  - (no client should be forced to depend on methods it does not use)
- D – Dependency inversion principle
  - (details should depend on abstraction, not the other way around)

REVIEW: WELL-DESIGNED SYSTEM: SOLID PRINCIPLES

Single responsibility – related to encapsulation

Open-closed – related to abstraction and inheritance

Liskov substitution – related to polymorphism

Interface segregation – related to encapsulation

Dependency inversion – related to abstraction

REVIEW: SOLID PRINCIPLES

Single responsibility: “A class should have one, and only one, reason to change.”

Open-closed: “Classes should be open for extension but closed for modification.”

Liskov substitution: “Derived classes must be substitutable for their base classes.”

Interface segregation: “Make fine-grained interfaces that are client specific.”

Dependency inversion: “Depend on abstractions, not on concretions.”

COLLABORATING WITH GITHUB

CS 5004, SPRING 2024 – LECTURE 8

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 before 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
```

IF YOU DON'T GIT PULL BEFORE MAKING CHANGES
INSTRUCTO
YOU'RE GONNA HAVE A BAD TIME

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 MAIN 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

CS 5004, SPRING 2024 – LECTURE 8

REVIEW: POLYMORPHISM

Polymorphism – the ability of one instance to be viewed/used as different types (the ability to take many shapes/forms/views)

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 **data 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

PARAMETRIC POLYMORPHISM

- Parametric polymorphism - ability for a function or type to be written such that it handles values identically without depending on knowledge of their types
  - Such a function or type is called a generic function or generic data type

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>();
```

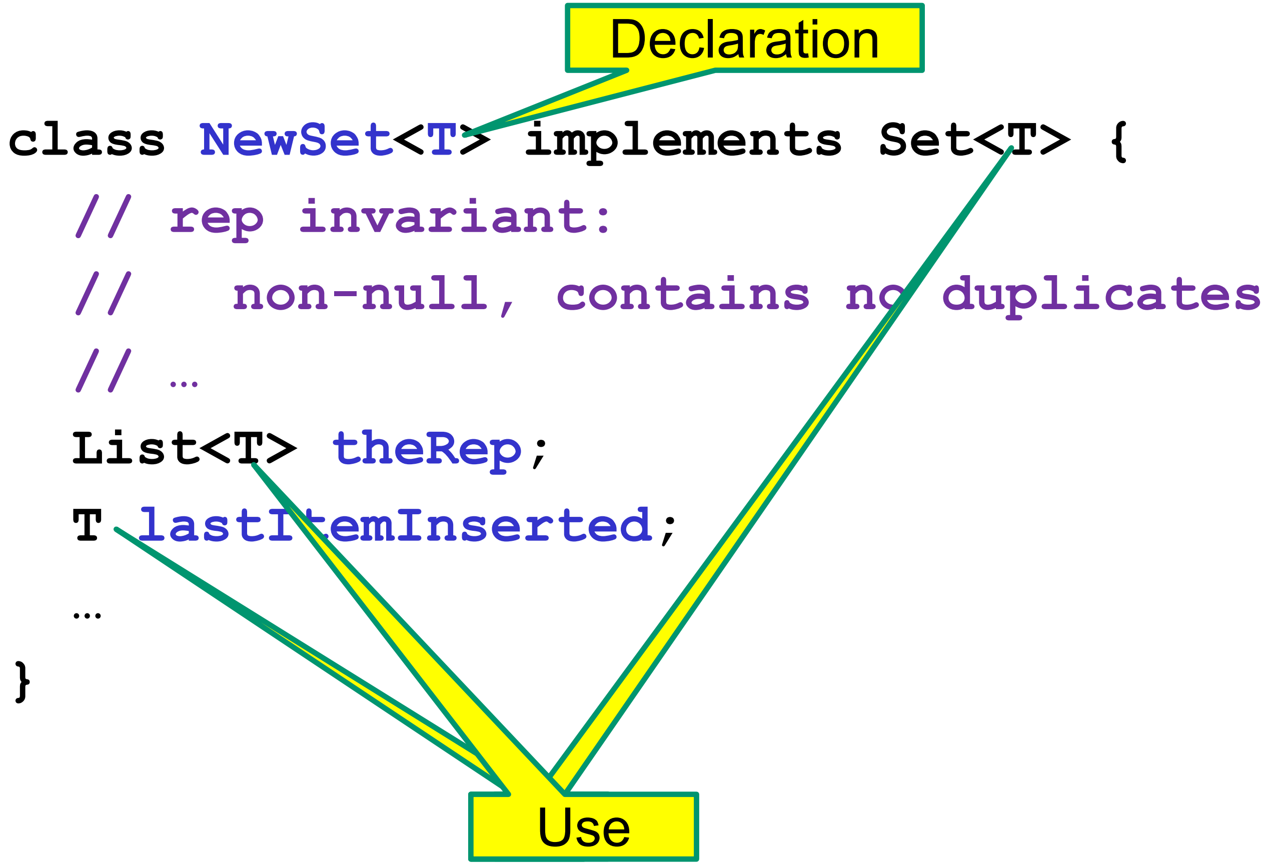
Instead, use the diamond operator, **<>**:

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

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

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

SUMMARY: TYPE VARIABLES ARE TYPES



Declaration

```
class NewSet<T> implements Set<T> {
  // rep invariant:
  //   non-null, contains no duplicates
  // …
  List<T> theRep;
  T lastItemInserted;
  …
}
```

Use

IMPLEMENTING GENERICS A.K.A. TYPE VARIABLES ARE TYPES

```
// a parameterized (generic) class
public class Name<Type> {…}
public class Name<Type, Type, ..., Type> {…}
interface Name<Type, Type, ..., Type> {…}
```

- By putting the Type in < >, we are demanding that any client that constructs our object must supply a type parameter
- We can require multiple type parameters separated by commas
- The convention is to use a 1-letter name:
  - T for Type
  - E for Element
  - N for Number
  - K for Key,
  - V for Value
- The type parameter is instantiated by the client (e.g., E → String)

GENERIC LINKED LIST ADT

CS 5004, SPRING 2024 – LECTURE 8

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

Use the placeholder anywhere you need to indicate type

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

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

EXAMPLE: VET CLINIC

CS 5004, SPRING 2024 – LECTURE 8

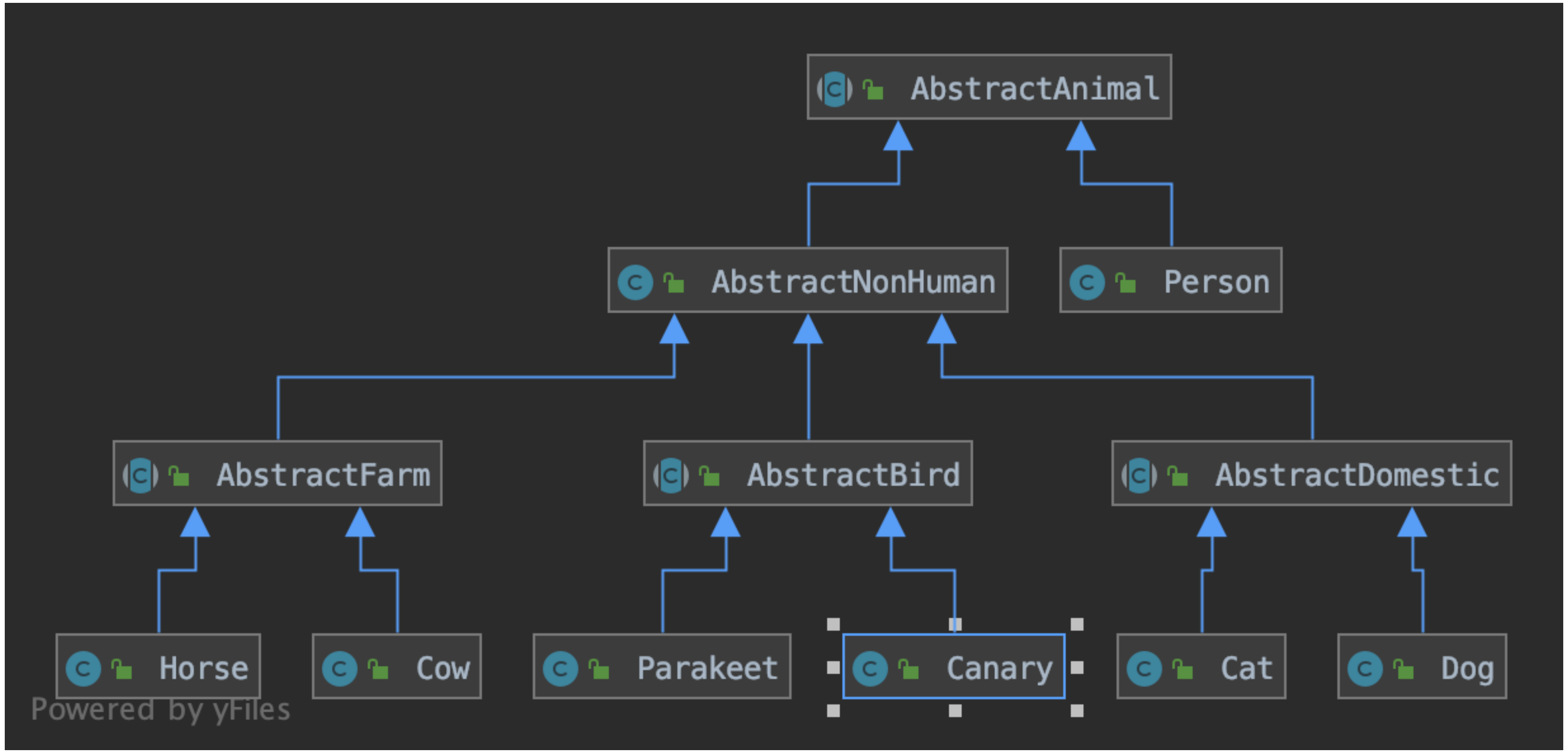
GENERIC CLASS FROM SCRATCH: VET CLINIC EXAMPLE

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

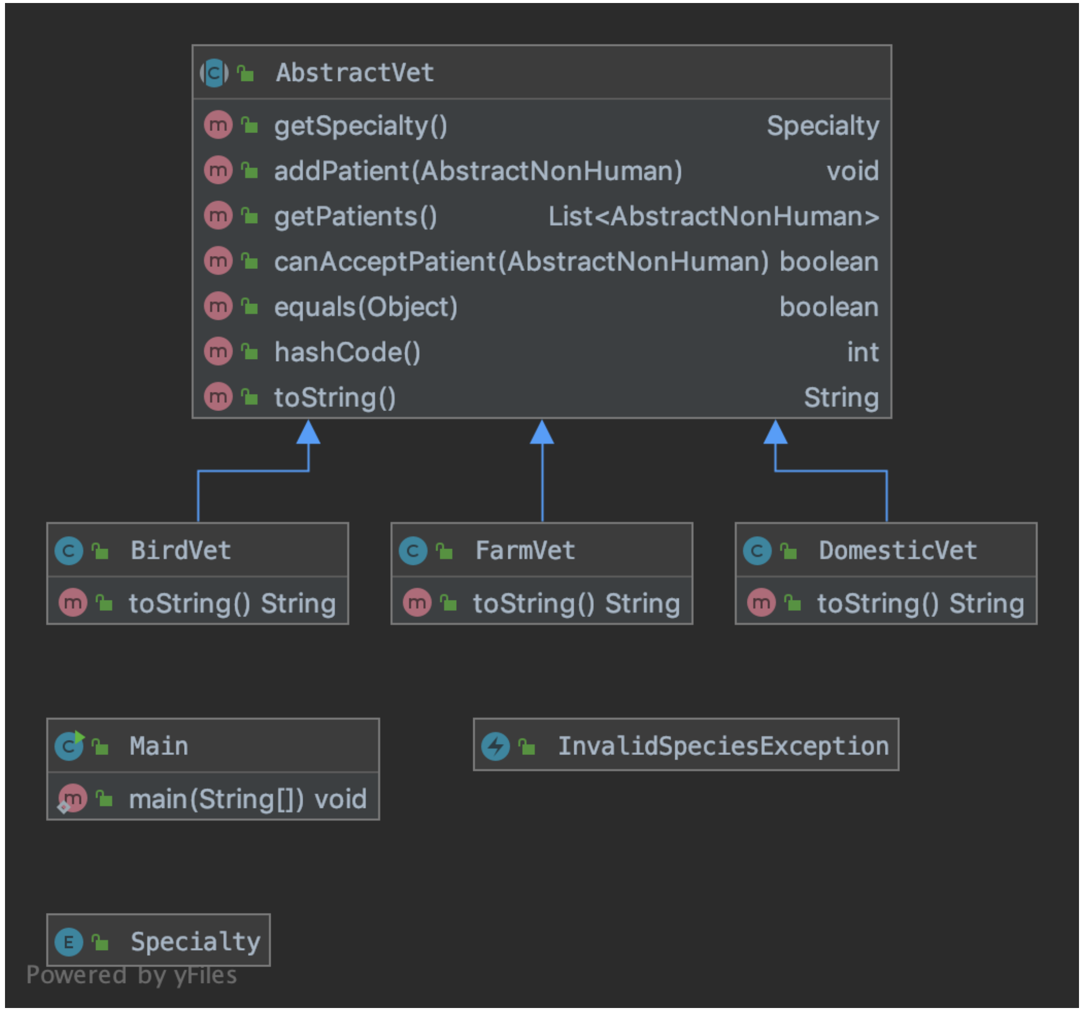
VET CLINIC EXAMPLE: ANIMALS



VET CLINIC EXAMPLE: USING INHERITANCE

In `AbstractVet`:

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



AbstractVet
getSpecialty() Specialty
addPatient(AbstractNonHuman) void
getPatients() List<AbstractNonHuman>
canAcceptPatient(AbstractNonHuman) boolean
equals(Object) boolean
hashCode() int
toString() String

BirdVet
toString() String

FarmVet
toString() String

DomesticVet
toString() String

Main
main(String[]) void

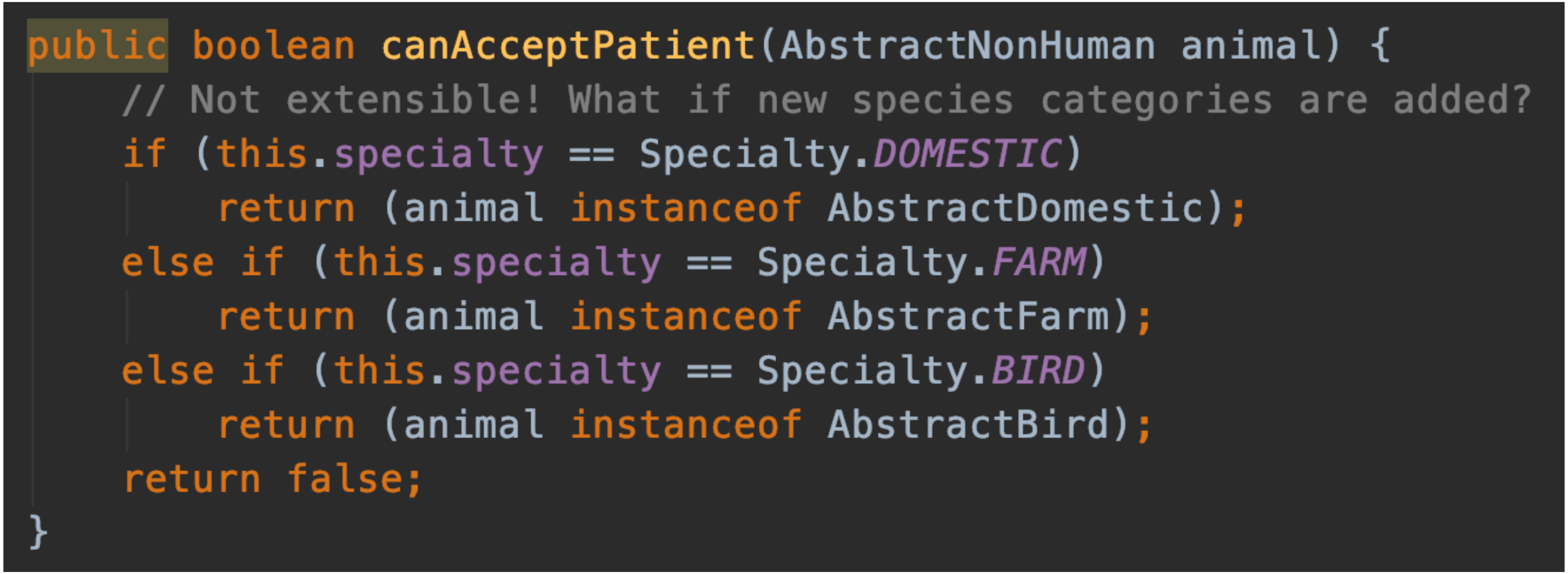
InvalidSpeciesException

Specialty

Powered by yFiles

VET CLINIC EXAMPLE: USING INHERITANCE

Adding a patient → must ensure the patient matches the specialty



```java
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

GENERICS – PART 2

CS 5004, SPRING 2024 – LECTURE 8

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

BOUNDED DATA PARAMETERS

CS 5004, SPRING 2024 – LECTURE 8

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.

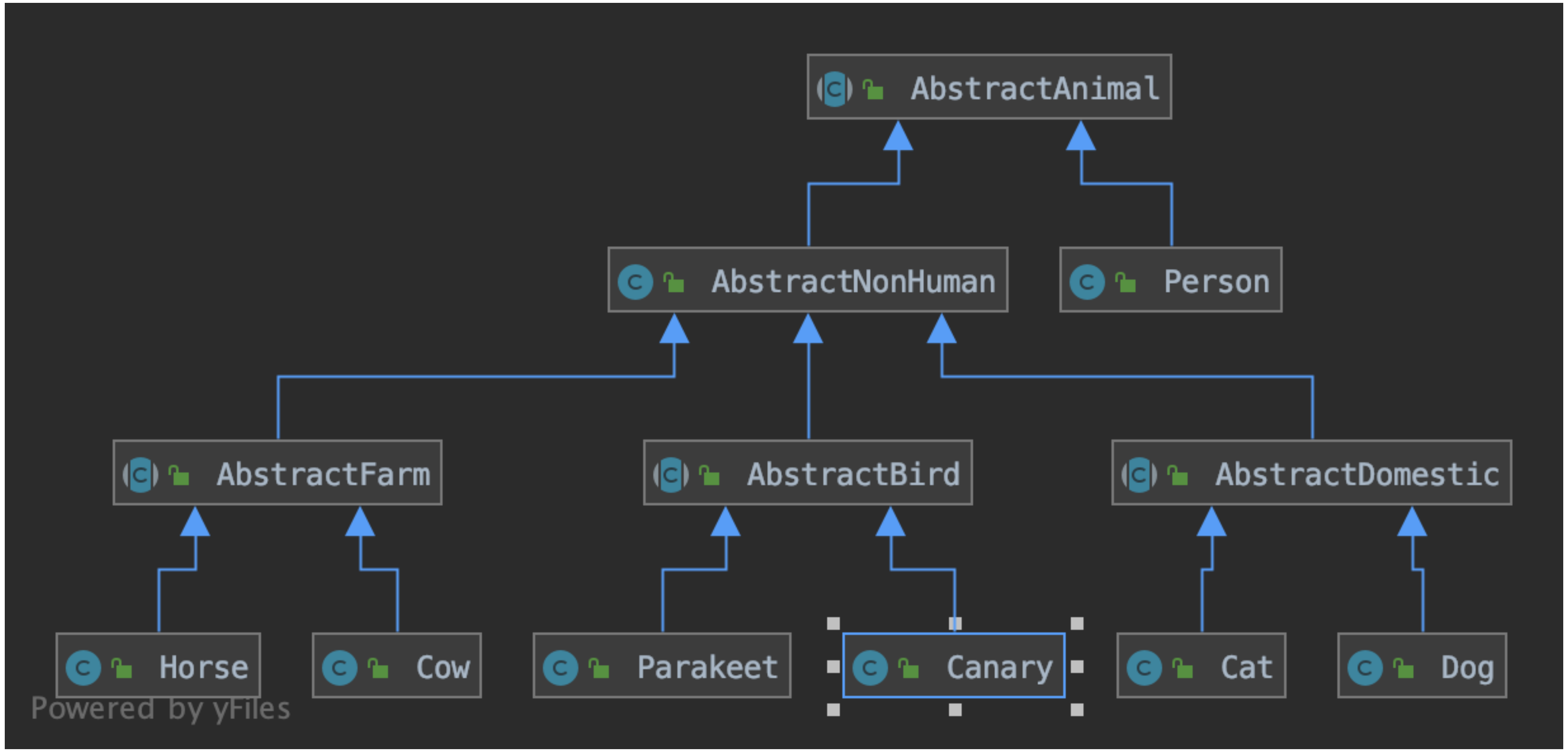
```
Person doolittle = new Person(“Dr.”, “Doolittle”);
Cat mittens = new Cat(“Mittens”, doolittle);

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!

GENERIC METHODS

CS 5004, SPRING 2024 – LECTURE 8

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>);`**
- The compiler will check that any params meet the placeholder needs:
  - Inherit Object if unbounded
  - Inherit the given class if bounded

STATIC METHODS WITH GENERICS

Sometimes it doesn’t make 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

CS 5004, SPRING 2024 – LECTURE 8

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

CS 5004, SPRING 2024 – LECTURE 8

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

GENERICS AND SUBTYPING

CS 5004, SPRING 2024 – LECTURE 8

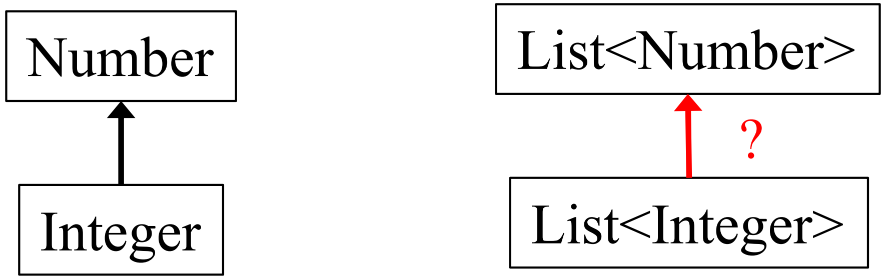
NOT ALL GENERICS ARE FOR COLLECTIONS

```
class Utils {
  static double sumList(List<Number> lst) {
      double result = 0.0;
      for (Number n : lst) {
        result += n.doubleValue();
      }
      return result;
  }
  static Number choose(List<Number> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

NOT ALL GENERICS ARE FOR COLLECTIONS

- Weaknesses:
- We would like to use `sumList` for any subtype of `Number`
  - For example, `Double` or `Integer`
- We would like to use `choose` for any element type
  - i.e., any subclass of `Object`
  - No need to restrict to subclasses of `Number`
  - Want to tell clients more about return type than `Object`
- Class `Utils` is not generic, but the methods should be generic

GENERICS AND SUBTYPING



- **`Integer`** is a subtype of **`Number`**
- Is **`List<Integer>`** a subtype of **`List<Number>`**?
- Use subtyping rules (stronger, weaker) to find out…

HARD TO REMEMBER?

If **`Type2`** and **`Type3`** are different,
then **`Type1<Type2>`** is *not* a subtype of **`Type1<Type3>`**

Previous example shows why:

- Observer method prevents "one direction"
- Mutator/producer method prevents "the other direction"

If our types have only observers or only mutators, then one direction of subtyping would be sound

- But Java's type system does not "notice this" so such subtyping is never allowed in Java

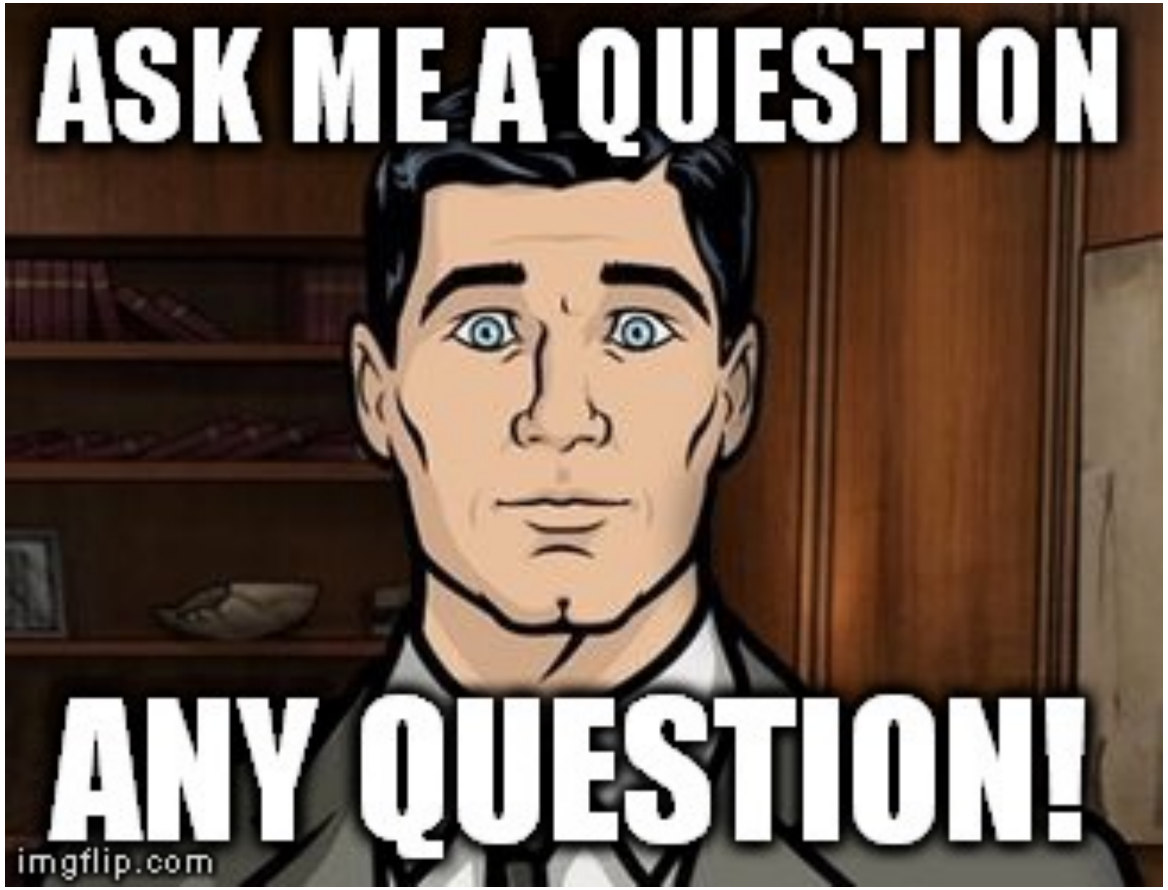
ABOUT PARAMETERS

- So we have seen **`List<Integer>`** and **`List<Number>`** are not subtype-related

- But there is subtyping "as expected" on the generic types themselves

- Example: If **`HeftyBag`** extends **`Bag`**, then
  - **`HeftyBag<Integer>`** is a subtype of **`Bag<Integer>`**
  - **`HeftyBag<Number>`** is a subtype of **`Bag<Number>`**
  - **`HeftyBag<String>`** is a subtype of **`Bag<String>`**
  - …

YOUR QUESTIONS



[Meme credit: imgflip.com]