CS5010: Lecture 5 Intro to Generics

Fall 2022

Brian Cross

Credits: Tamara Bonaci, Tony Mullen, Maria Zontak

Homework Tips

- Commit your project
  - We will build it
    - Please ensure that it will work in a new clone to run “doAll” task in build.gradle
  - We will run your tests (via doAll)
    - No need to commit generated reports
  - We will generate Javadoc (via doAll)
    - No need to commit generated documentation.
- Once your Pull Request is approved, you can complete and merge into **main**
  - Will block PRs that have more than intended topic
- Want help?
  - Office hours
  - Piazza (public or private)
  - Ask early
- Commit early and often
  - Then push to server

Administrative Bits

- UML Predesigns
  - were due Monday
  - Feedback is out
- Homework #2 due Monday @ 11:59pm
  - Did you start? ☺
- Lab 3 due Friday @ 11:59pm
- Homework #1 grades out this week

Administrative Bits

- No Lab next week
- Codewalk #2
  - Can be done from home.
  - Details to come out Monday
  - Due Wednesday @ 11:59pm

Agenda

- Polymorphism recap
- Generics
- Interface **`Comparable`**

Polymorphism

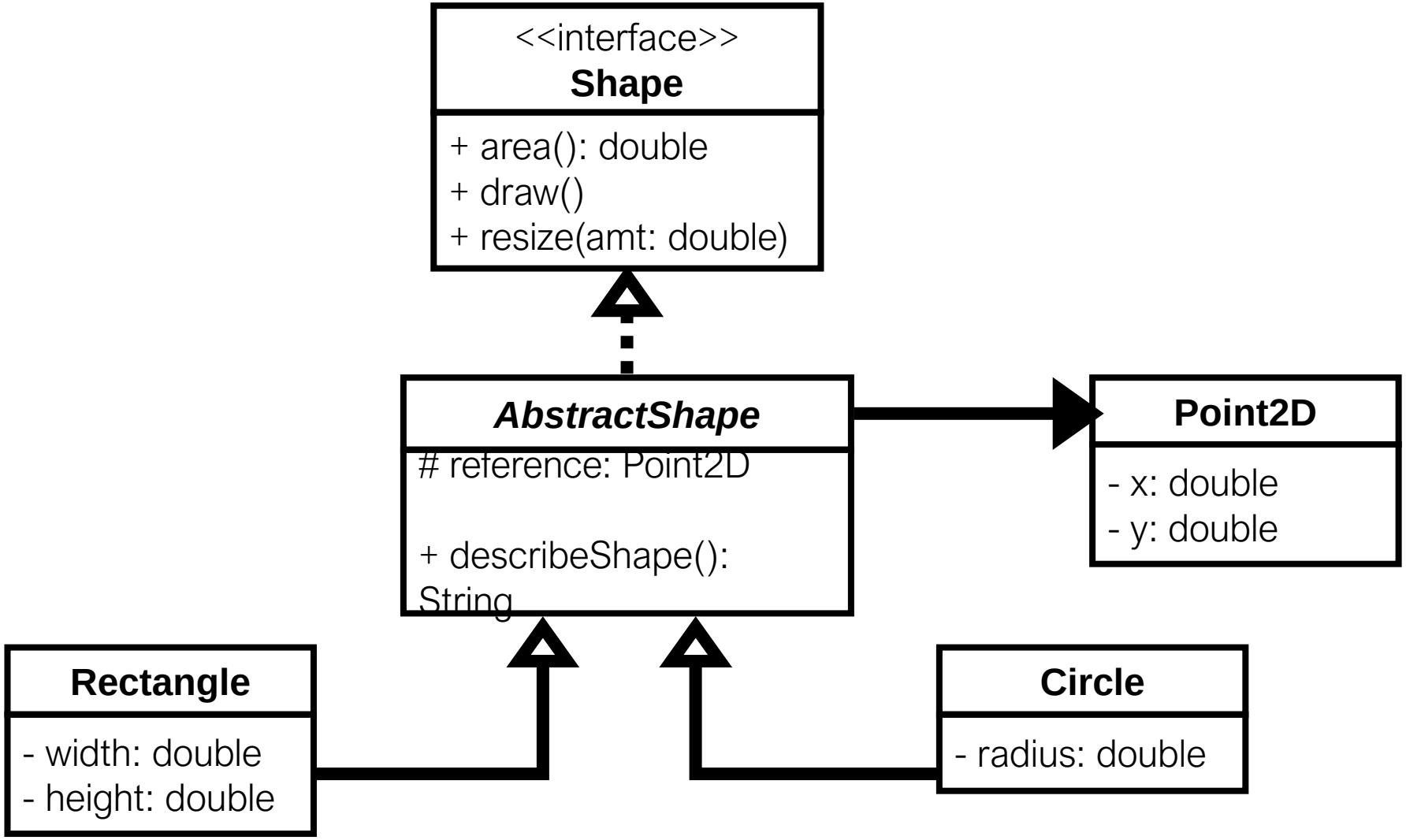
A recap

Polymorphism basics

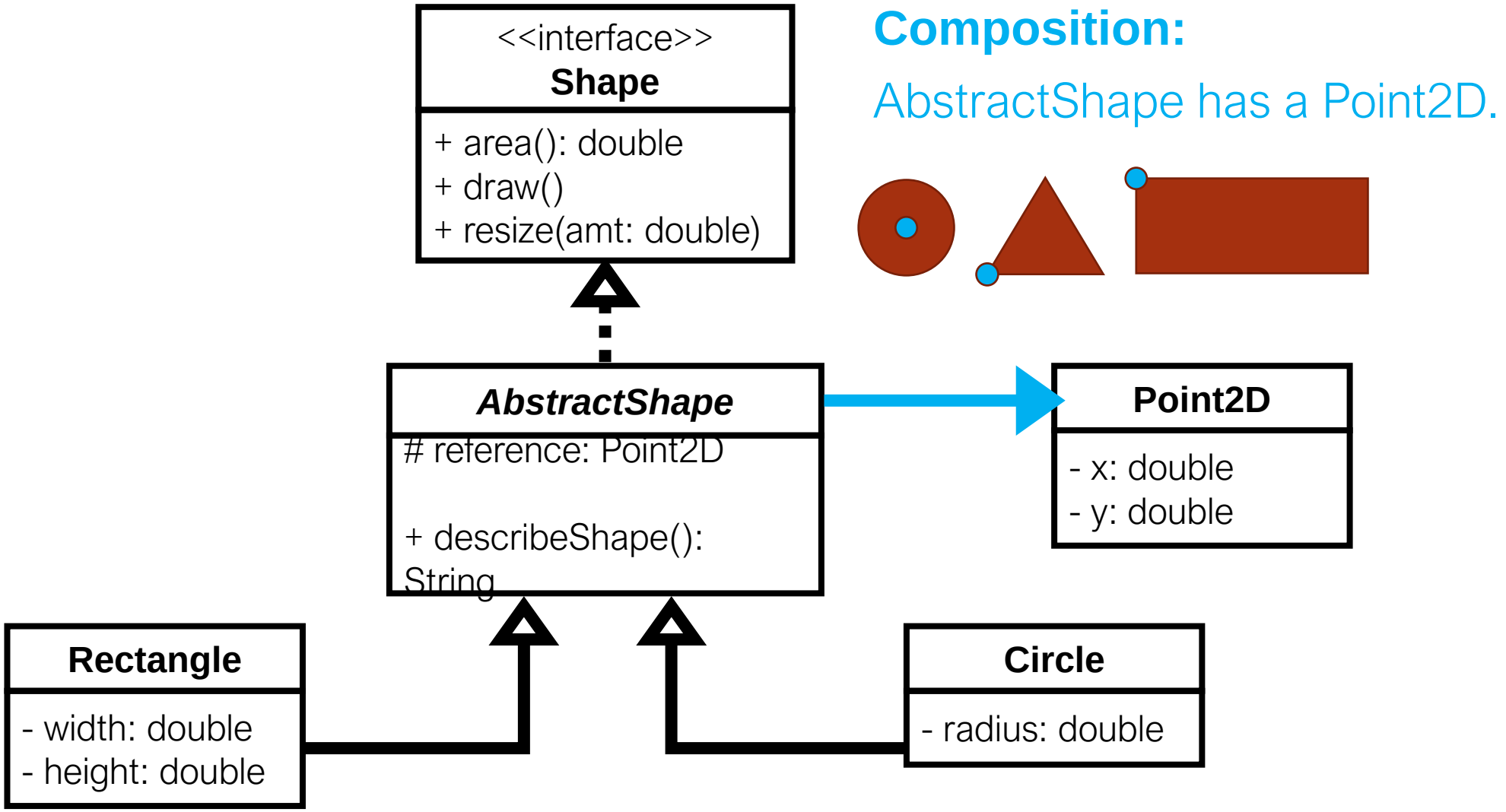
Polymorphism:

The ability of one instance to be viewed/used as different types.

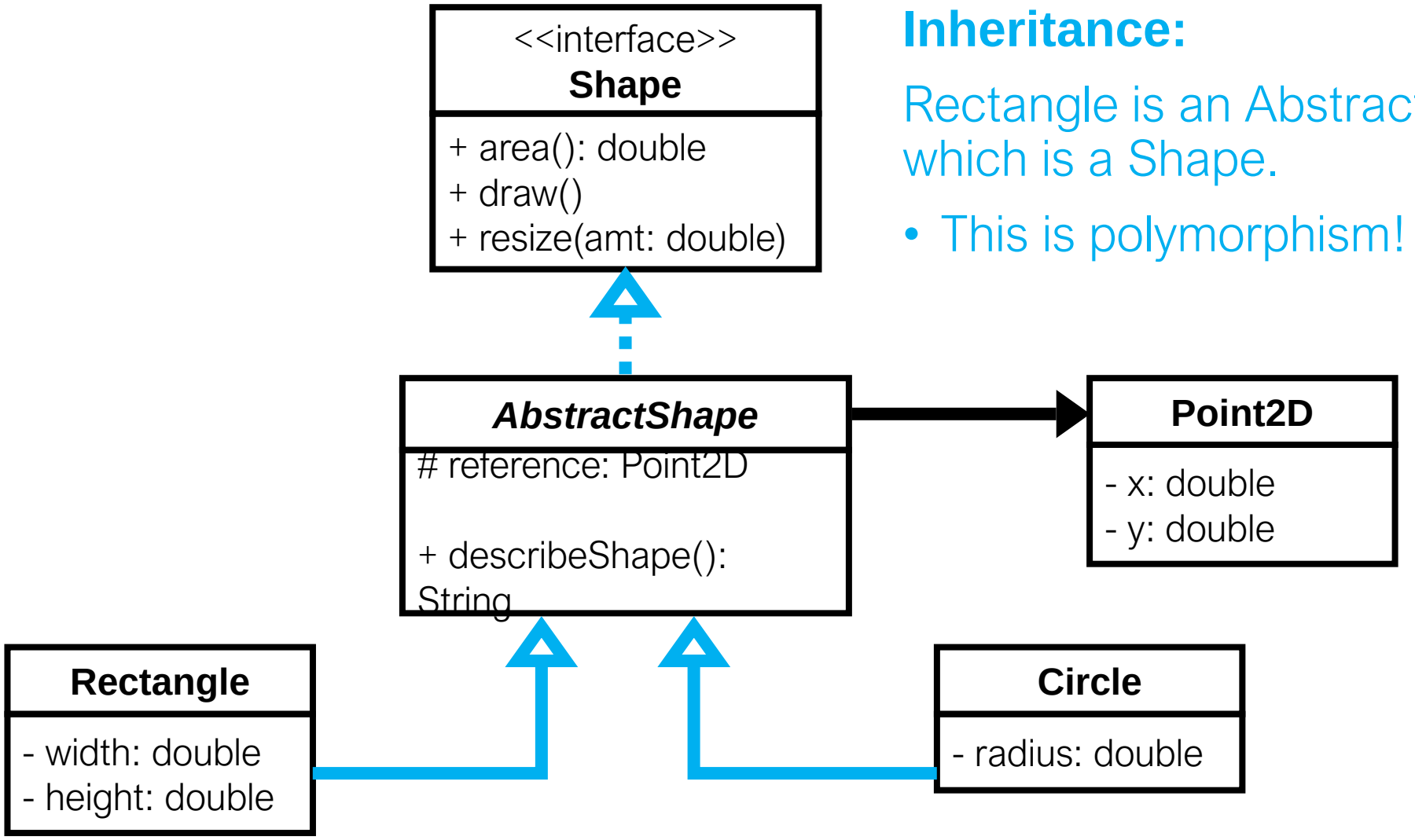
Shapes package



Shapes package



Shapes package



Inheritance:

Rectangle is an AbstractShape, which is a Shape.

- This is polymorphism!

Polymorphism example

```
Point2D pt = new Point2D(0, 0);

Circle circle = new Circle(pt, 5);

Rectangle square
       = new Rectangle(pt, 10, 10);

Rectangle rect
       = new Rectangle(pt, 5, 10);
```

Polymorphism example

```
Point2D pt = new Point2D(0, 0);

Circle circle = new Circle(pt, 5);

Rectangle square
       = new Rectangle(pt, 10, 10);

Rectangle rect
       = new Rectangle(pt, 5, 10);
```

```
Point2D pt = new Point2D(0, 0);

Shape circle = new Circle(pt, 5);

Shape square
       = new Rectangle(pt, 10, 10);

AbstractShape rect
       = new Rectangle(pt, 5, 10);
```

Polymorphism basics

Polymorphism:

The ability of one instance to be viewed/used as different types.

- Useful when we want to write code that can handle all subclasses at once.
  - E.g. a method to align all shapes by their reference points → don't care if the shape is a circle or square

Another form of polymorphism: Overloading

All classes extending **`AbstractShape`** must supply a reference point to the constructor

- Could we have a “default” reference point?

`Rectangle` must be instantiated with a width and height

- Could we write a shortcut to create a square?

```
protected AbstractShape(Point2D reference) {
  this.reference = reference;
}
```

```
public Rectangle(Point2D reference, double width, double height) {
  super(reference);
  this.width = width;
  this.height = height;
}
```

Constructor overloading

A single class can have multiple constructors:

- Each constructor takes a different number or type of arguments.
- When an instance of the class is created, the constructor that matches the provided parameters will be called

Constructor overloading

```
public abstract class AbstractShape implements Shape {
  public Point2D reference;

  protected AbstractShape() {
    this.reference = new Point2D(0,0);
  }

  protected AbstractShape(Point2D reference) {
    this.reference = reference;
  }
```

Method overloading

```
public int fooBar(int a) {
  return a * 3;
}

public int fooBar(int a, int b) {
  return a * b;
}

public String fooBar(String a, String b)
{
  return a + b;
}
```

Java will match the method based on the parameters passed.

```
my_var.fooBar(3);

my_var.fooBar(2, 4);

my_var.fooBar("Hello", "World");
```

Method overloading rules

Two or more methods are overloaded if:

- The method name is the same
- The argument list differs in:
  - The number of arguments
  - The types of the arguments
  - The order of argument types

Caution:

- Multiple methods with identical signatures won't compile
- Return types don't count in overloading

What exactly is being polymorphic?

So far:

- Objects
  - Instance of subclass (e.g. Square) treated as instance of super class (e.g. Shape)
- Methods – overloading

Generics

AKA 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.

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<>();
```

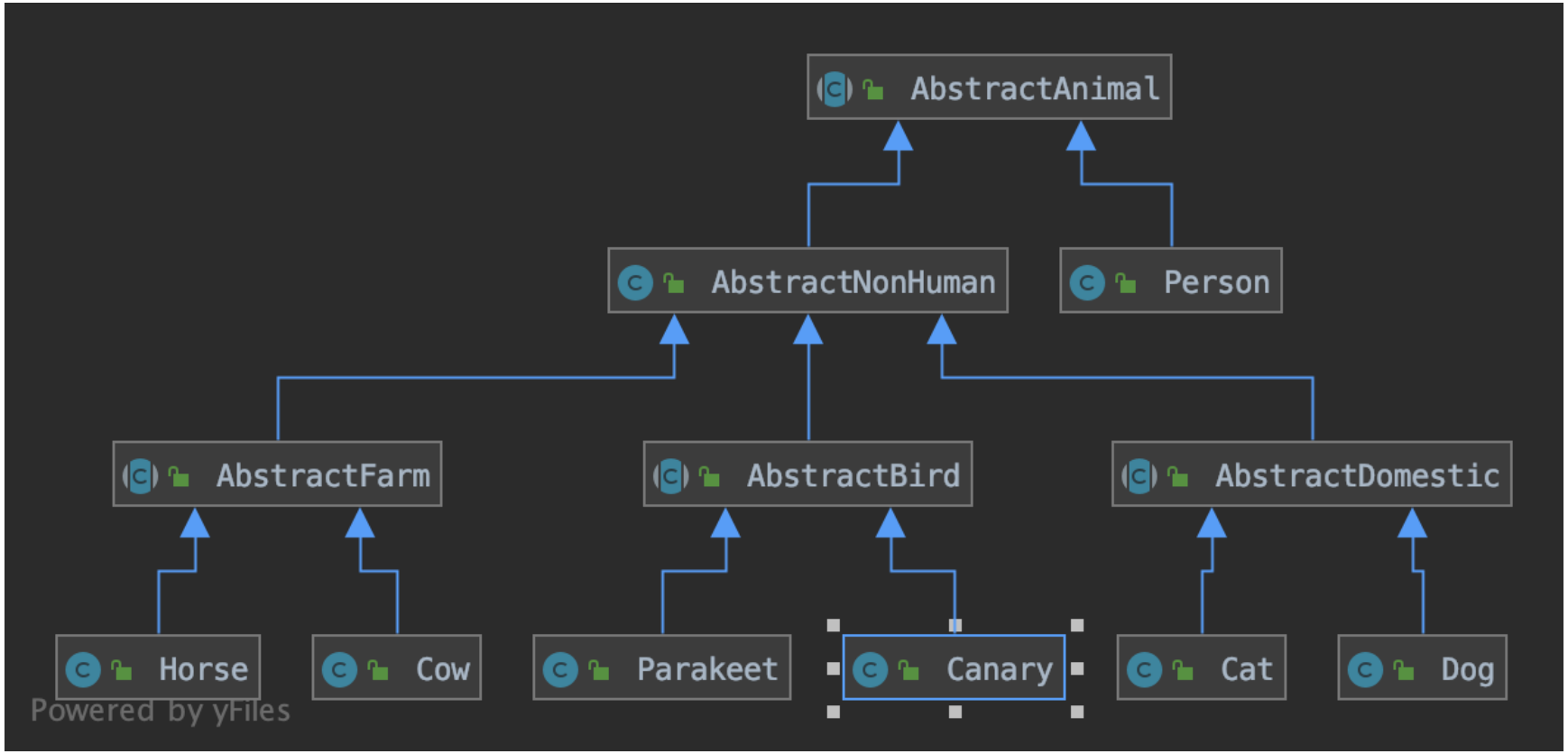
Writing a generic class: Vet clinic example

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

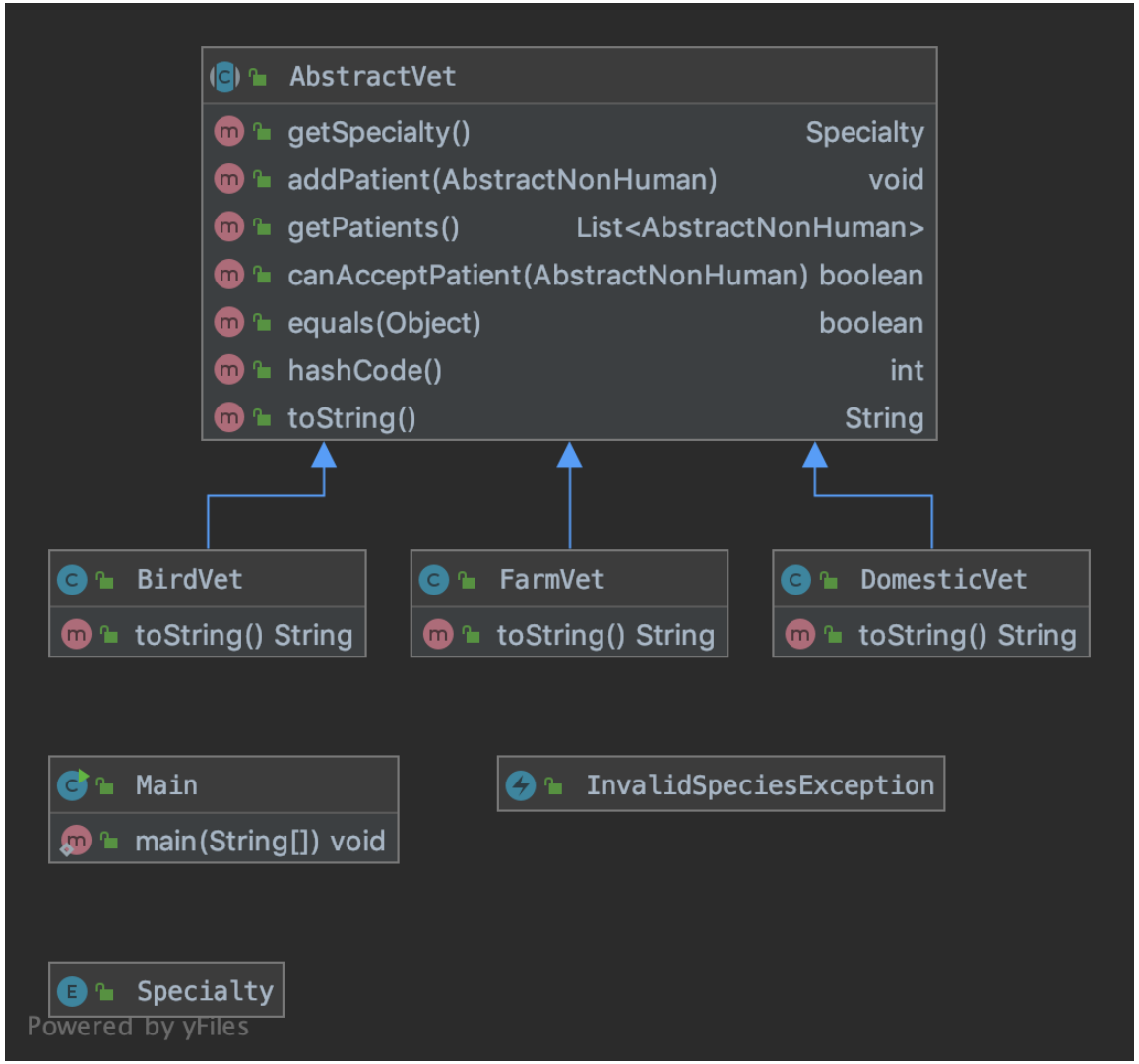
Vet clinic example: animals



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

`<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

Using a generic type

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

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

Using a generic type

Specify `T` when declaring and instantiating:

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

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

Guaranteeing type safety

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

PatientList<Cat> catsOnly = new PatientList<>(100);

catsOnly.addPatient(mittens);
catsOnly.addPatient(spot);
```

→ **compile-time error**

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

Try it

Write a HighScore class that extends the Pair class

HighScore contains:

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

Implementing a generic interface

```
public interface IListADT<T> {
  IListADT append(T item);
  IListADT insert(int index, T item) throws IndexOutOfBoundsException;
  T itemAt(int index) throws IndexOutOfBoundsException;
  int size();
  IListADT remove(T item) throws ItemNotFoundException;
}
```

Implementing a generic interface

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

```
IListADT append(Object item)
{
   …
}
Object itemAt(int index) {
   …
}
```

Implementing a generic interface

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

- **A problem for client code**
- Will not enforce type requirements → runtime errors that are hard to anticipate.

```
IListADT append(Object item)
{
   …
}

Object itemAt(int index) {
   …
}
```

Converting from Object to generic <T>

```
public class GenericEmptyNode implements IListADT {
  IListADT append(Object item)…
  IListADT insert(int index, Object item)…
  Object itemAt(int index)…
  int size()…
  IListADT remove(Object item)…
}
```

Converting from Object to generic <T>

```
public class GenericEmptyNode<T> implements IListADT<T> {
  IListADT append(Object item)…
  IListADT insert(int index, Object item)…
  Object itemAt(int index)…
  int size()…
  IListADT remove(Object item)…
}
```

Change the header to indicate this class takes generic parameters

Converting from Object to generic <T>

```
public class GenericEmptyNode<T> implements IListADT<T> {
   IListADT<T> append(Object item)…
   IListADT<T> insert(int index, Object item)…
   Object itemAt(int index)…
   int size()…
   IListADT<T> remove(Object item)…
}
```

Add <T> to all references to the interface

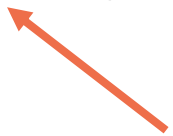
Converting from Object to generic <T>

```
public class GenericEmptyNode<T> implements IListADT<T> {
   IListADT<T> append(T item)…
   IListADT<T> insert(int index, T item)…
   T itemAt(int index)…
   int size()…
   IListADT<T> remove(T item)…
}
```

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

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

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


Break time

Generic methods

- Allow you to write one method that can handle different argument types
- Can (sometimes) be used instead of method overloading

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

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

? 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 – method

`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 – client method

`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

is 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 can 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
```

Extension: doctors and vets

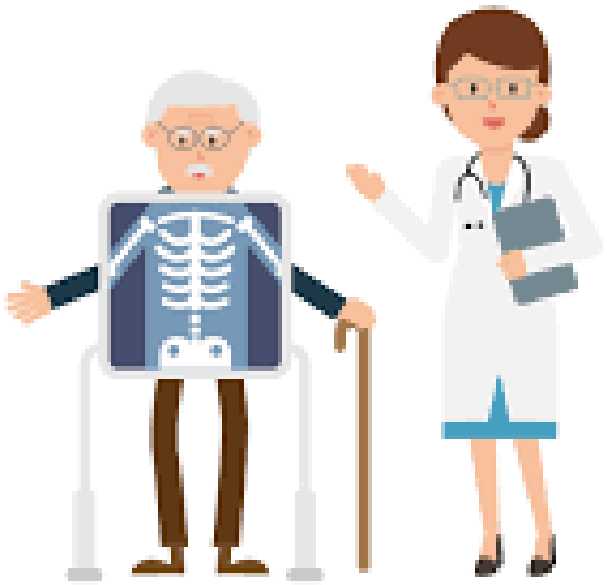
Now the company wants to support (human) doctors too!

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

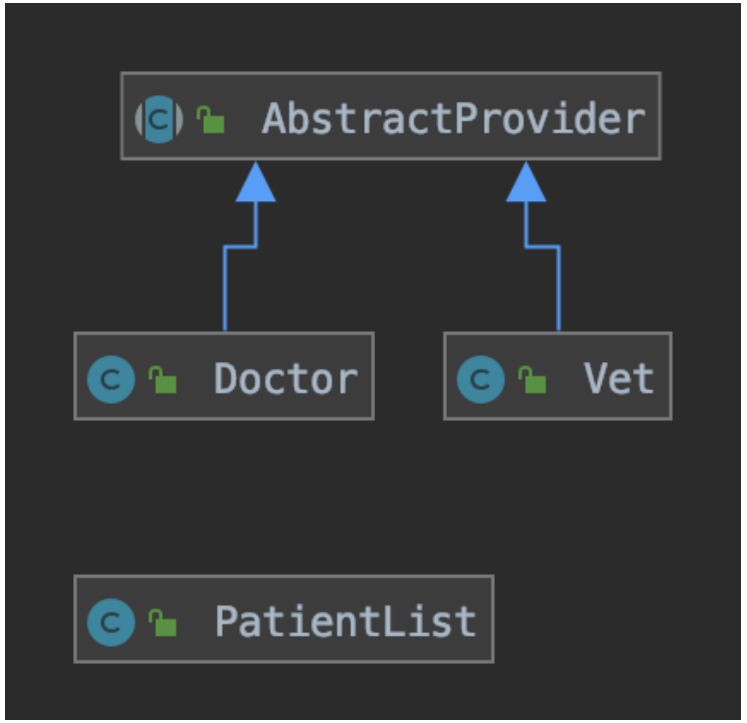


Try it: doctors and vets

Refactor the code in GenericPatientList to work for doctors and patients alike → Use generics!

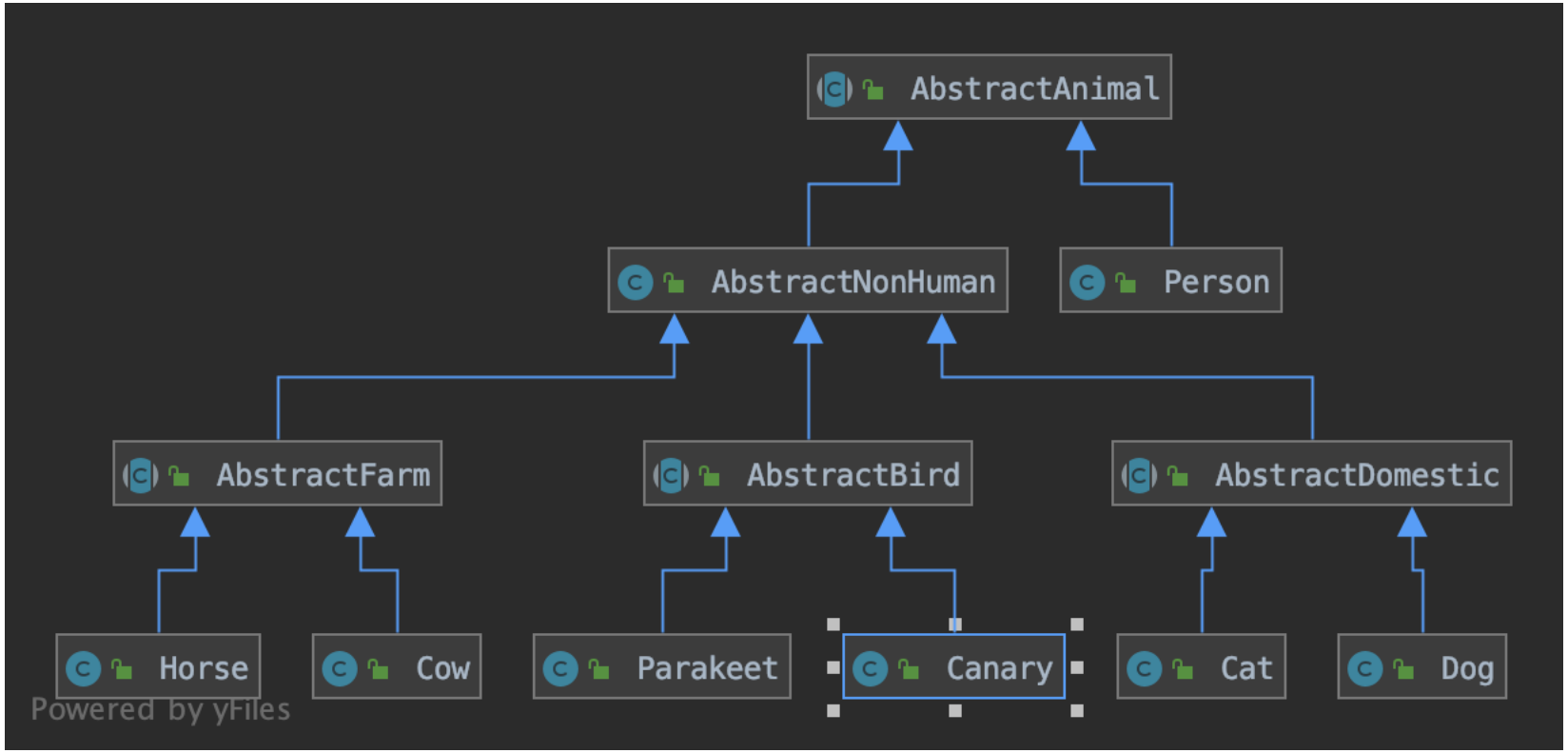
Key difference:

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



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!

Interface **Comparable**

Comparing objects

We already know how to determine if two objects are **equal**

```
myObject.equals(otherObject);
```

- Override Java's built-in `equals` method to enable checking if two objects are the same based on their properties

Comparing objects

What if we want to know if `myObject` is less than/greater than `otherObject`?

- Why might we want to know this?

Comparing objects

- What if we want to know if `myObject` is less than/greater than `otherObject`?
  - Why might we want to know this?
  - Sorting collections of Objects… or other purposes:
    - e.g. two playing cards – which is worth more/less in a game

Comparing objects with `compareTo`

Many of Java's built-in objects (e.g. String, Integer) can be compared using `compareTo`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ output?

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`
- Returns **postive int** if `this` is greater than `other` e.g. `abc > lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **output?**

Comparing objects with `compareTo`

- Returns **negative int** if `this` is less than `other` e.g. `abc < lower`
- Returns **zero** if `this` equals `other` e.g. `abc.equals(lower)`
- Returns **postive int** if `this` is greater than `other` e.g. `abc > lower`

```
String abc = "ABC";
String lower = "abc";
abc.compareTo(lower);
```

→ **-32: abc < lower**

Comparing objects with `compareTo`

Can be used as a test in if statements e.g.

```
if (abc.compareTo(lower) < 0) {
  // Do this if abc comes before lower
}
```

Implementing `compareTo` in your classes

Unlike `equals` & `hashCode`, there is no `compareTo` to override by default

Implementing `compareTo` in your classes

Unlike `equals` & `hashCode`, there is no `compareTo` to override by default

- Implement `Comparable<T>`
- Replace `T` with the class name

```
public class MyClass implements
                    Comparable<MyClass> {
    ...
}
```

Implementing `compareTo` in your classes

Implement `Comparable<T>`'s required method, `compareTo`

- If your header is correct (you changed T to the class name), IntelliJ will generate the method skeleton as shown
  - Otherwise `o` will be an `Object`

```
public class MyClass implements
                Comparable<MyClass> {

  public int compareTo(MyClass o) {

    …

  }

}
```

Implementing `compareTo` in your classes

Return:

- Negative (usually -1) if this is less than o
- Zero if this equals o
- Positive (usually 1) if this is greater than o

```
public class MyClass implements
                Comparable<MyClass> {

  public int compareTo(MyClass o) {

    …

  }

}
```

Rules for implementing `compareTo`

- If A is less than B, then B must be greater than A
- If A is less than B and B is less than C, then A must be less than C
- If A is not less than B and B is not less than A, then A and B must be equal
  - If `A.equals(B)` is `true`, `A.compareTo(B)` should return 0.

Exercise: making `Birthday` `Comparable`

- Take a look at the Birthday class:
- How would you implement `compareTo`?:
  - When should a Birthday be **less than** another Birthday?
  - When should a Birthday be **equal to** another Birthday?
  - When should a Birthday be **greater than** another Birthday?

```
public class Birthday
    extends Triple<Integer, Integer, Integer>
    implements Comparable<Birthday> {

    public Birthday(Integer month,Integer day,
        Integer year) {...}

    public Integer getMonth() {...}

    public Integer getDay() {...}

    public Integer getYear() {...}

    public boolean equals(Object o) {...}

    public int compareTo(Birthday o) {



    }
}
```

Making `Birthday` `Comparable`

One possible solution:

- Check `equals` first
- Then, use `compareTo` on the years
- If years are equal, use month as tie breaker
- If months are equal, use day as tie breaker

Summary

What we covered

Generics (parametric polymorphism):

- Enable **types** to be **parameters** when defining classes and interfaces.
- Writing and using generic classes and methods
- Bounding generics
- Wildcards
- Type erasure

Interface `Comparable`:

- Is one Object less than, equal to, or greater than another?

Code from tonight's lecture

https://github.khoury.northeastern.edu/cs5010seaF22/Code_From_Lectures/Evening_Lectures/Lecture5