CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 5

Divya Chaudhary
d.chaudhary@northeastern.edu

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Review
  - Abstract Data Type (ADT)
  - Introduction to polymorphism
  - Subtype polymorphism
  - Static and dynamic polymorphism
  - Casting
  - Ad hoc polymorphism

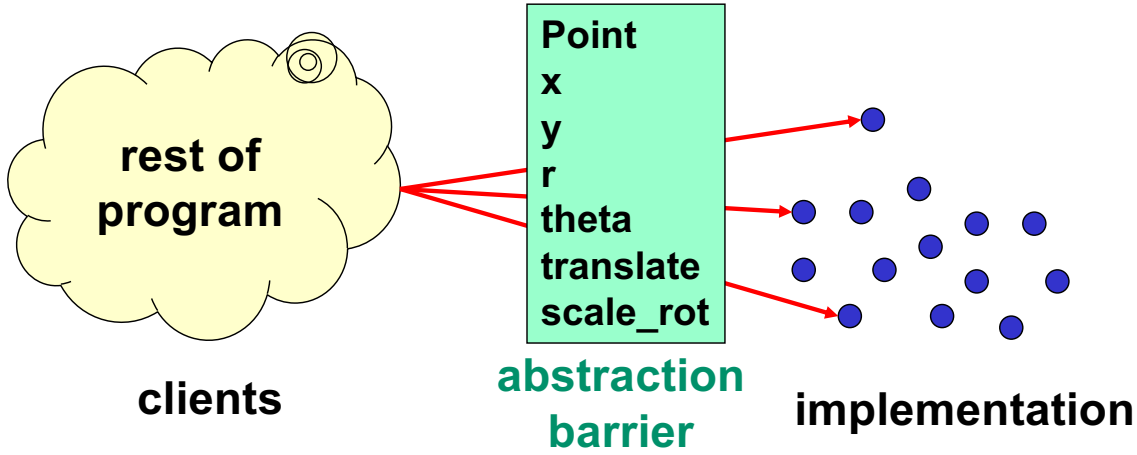
REVIEW

CS 5004, SPRING 2022– LECTURE 5

REVIEW: ABSTRACT DATA TYPE

- Abstract Data Type (ADT) - model that describes data by specifying the operations that we can perform on them
- Clients care about the ADT
- For each operation, we describe:
  - The expected inputs, and any conditions that need to hold for our inputs and/or our ADT
  - The expected outputs and any conditions that need to hold for our output and/or our ADT
  - Invariants about our ADT

REVIEW: ADT = OBJECT + OPERATIONS



- Implementation is hidden
- The only operations on objects of the type are those provided by the abstraction

REVIEW: SPECIFYING A DATA ABSTRACTION

- A collection of procedural abstractions
  - Not a collection of procedures

- An abstract state
  - Not the (concrete) representation in terms of fields, objects, …
  - "Does not exists", but used to specify operations
  - Concrete state, not part of the specification, implements the abstract state

- Each operation described in terms of "creating", "observing", "producing" or "mutating"
  - No operations other than those in the specification

REVIEW: SPECIFYING AN ADT

Specification must include:

- **Overview**
- Abstract state
- Operation specifications

A description of what the class is for. Also specify whether it's mutable / immutable.

REVIEW: SPECIFYING AN ADT

Specification must include:

- Overview
- **Abstract state**
- Operation specifications

Describe key fields without implementation details. E.g. a “Person has a first and last name.”

REVIEW: SPECIFYING AN ADT

Specification must include:

- Overview
- Abstract state
- **Operation specifications**

Specifications of every operation (i.e. method) to be supported by the ADT.

REVIEW: SPECIFYING AN ADT

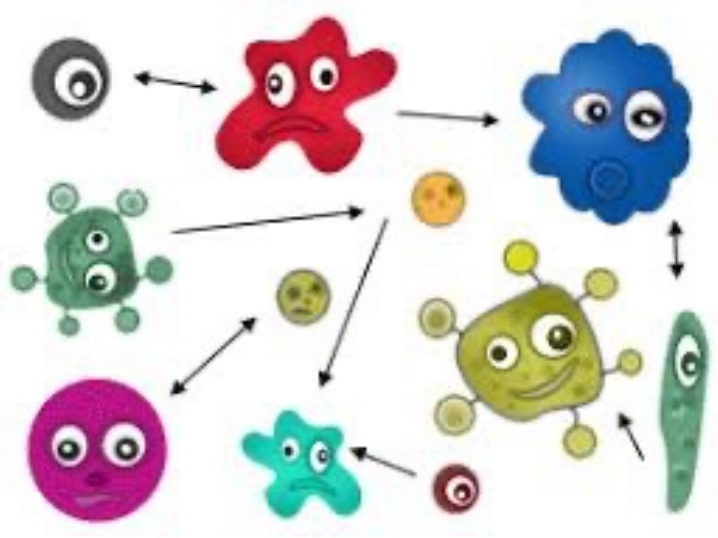
Immutable	Mutable
1. overview	1. overview
2. abstract state	2. abstract state
3. creators	3. creators
4. observers	4. observers
5. producers	5. producers (rare)
~~6. mutators~~	6. mutators

- Creators: return new ADT values (e.g., Java constructors)
- Producers: ADT operations that return new ADT values
- Mutators: modify a value of an ADT
- Observers: return information about an ADT

INTRODUCTION TO POLYMORPHISM

CS 5004, SPRING 2022– LECTURE 5

INTRODUCTION TO POLYMORPHISM



[Pictures credit: http://www.thewindowsclub.com/polymorphic-virus]

Polymorphism – the ability to define different classes and methods as having the same name but taking different data types

POLYMORPHISM DEFINITION

Having "many forms"

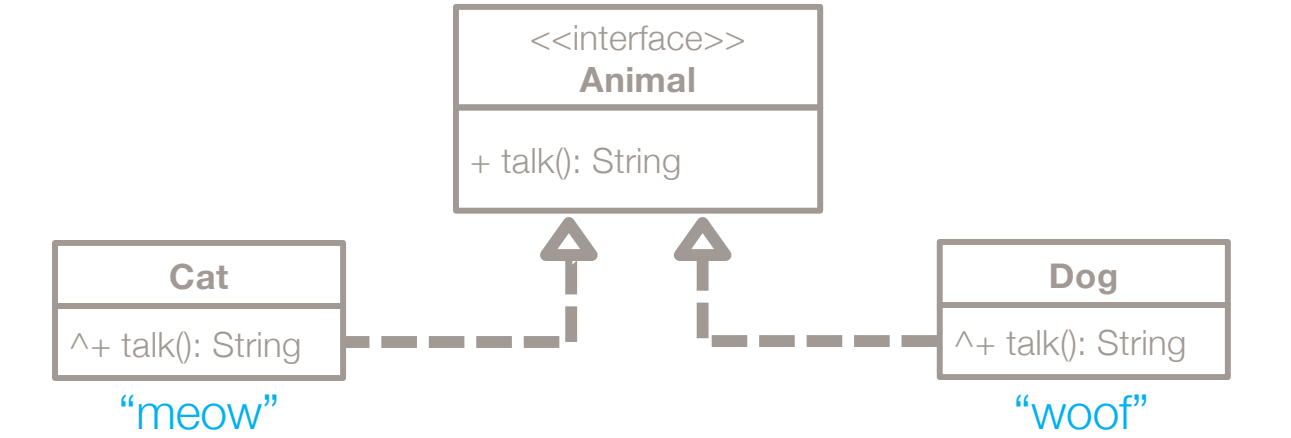
- Static / compile-time
  - Method / constructor overloading
- Dynamic / runtime
  - Subtype polymorphism – a result of inheritance
  - Includes method overriding

EXAMPLE: OVERLOADING

```java
/**
 * Helper function used by add and insert methods. Copies the items from one array to another.
 * @param from The array to copy from.
 * @param to The array to copy to.
 * @param fromStart The index in the "from" array to start copying from.
 * @param fromEnd The index in the "from" array to end at (exclusive).
 * @param toStart The index in the "to" array to start copying to.
 */
private void copyItems(String[] from, String[] to, int fromStart, int fromEnd, int toStart) {
    if (toStart >= 0) {
        for (int i = fromStart; i < fromEnd; i++) {
            to[toStart] = from[i];
            toStart++;
        }
    }
}

/**
 * Shortcut version of the helper method above. Will copy the entirety of the "from" array to the "to" array.
 * @param from The array to copy from.
 * @param to The array to copy to.
 */
private void copyItems(String[] from, String[] to) {
    this.copyItems(from, to, fromStart: 0, from.length, toStart: 0);
}
```

EXAMPLE: SUBTYPE POLYMORPHISM



POLYMORPHISM DEFINITION

Having "many forms"

- Static / compile-time
  - Method / constructor overloading
- Dynamic / runtime
  - Subtype polymorphism – a result of inheritance
  - Includes method overriding

Advantages

- Provides flexibility
- Reduces code duplication

POLYMORPHISM DEFINITION

Having "many forms"

- Static / compile-time
  - Method / constructor overloading
- Dynamic / runtime
  - Subtype polymorphism – a result of inheritance
  - Includes method overriding

?

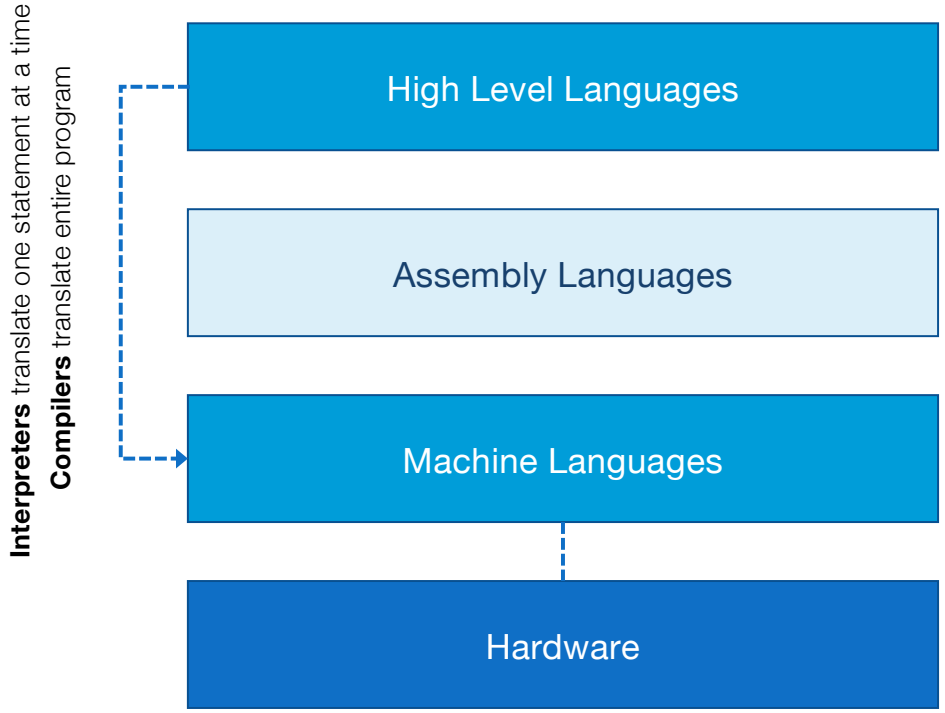
Advantages

- Provides flexibility
- Reduces code duplication

POLYMORPHISM

- Static binding/Compile-Time binding/Early binding/Method overloading.
  - (in same class)
- Dynamic binding/Run-Time binding/Late binding/Method overriding.
  - (in different classes)
-

PROGRAMMING LANGUAGES

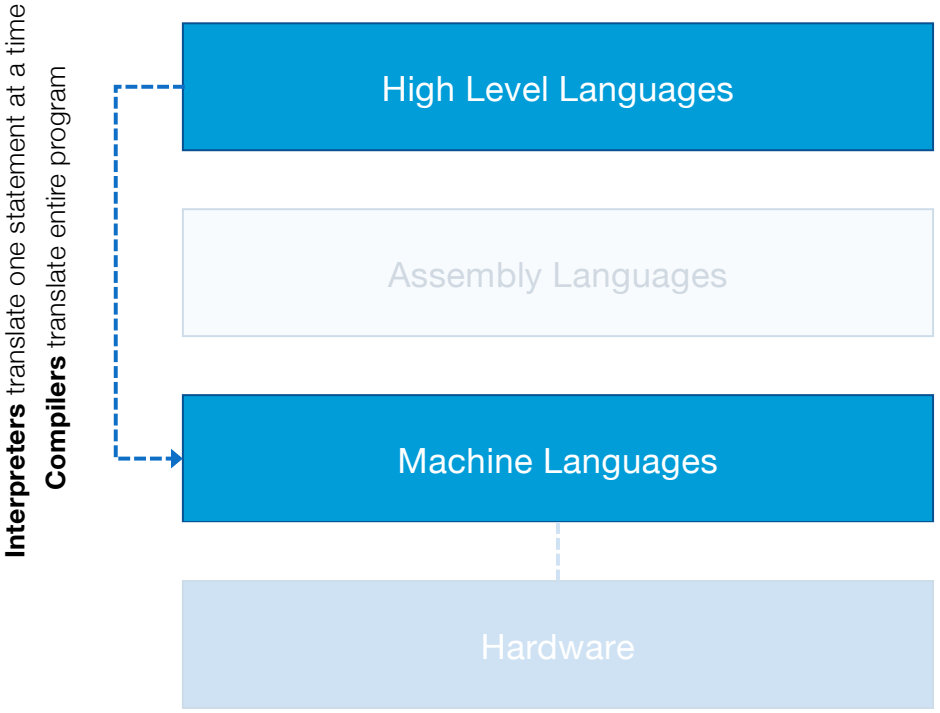


For humans.
Require translation to machine language.

Slightly more human-friendly than machine language. Specific to hardware & OS.

The only language understood by the machine.
Binary i.e., 0s and 1s

PROGRAMMING LANGUAGES



Python & Java are both high level languages

- Python has an interpreter
- Java has a compiler

The only language understood by the machine.
Binary i.e., 0s and 1s

COMPILE TIME VS. RUN TIME

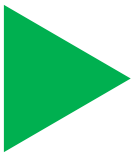
Compile time

- Refers to the process of translating your Java source code into machine language that your computer can run.

Run time

- When your compiled code actually runs, either in the IDE or as a Java application.

THE LITTLE GREEN BUTTON IN INTELLIJ…



1. Compiles any new code written since the last run
2. Runs the compiled code

COMPILE-TIME VS. RUNTIME ERRORS

Compile time

Errors detected when the code is being compiled.

- E.g., syntax errors
- Code with compile time errors won't finish compiling and therefore won't run

Runtime

Errors that happen while your code is actually running.

- Often due to unexpected input.
- When an app crashes, you've experienced a run-time error.

STATIC/ COMPILE TIME POLYMORPHISM

CS 5004, SPRING 2022– LECTURE 5

OVERLOADING

When a constructor or method has multiple implementations

- Each overloaded constructor / method *must* have different parameters
  - Number of parameters and/or type of parameters
- Return type can vary if parameters are unique

Why do this?

- Allows for "default" values
- Perform the same operations for different types
  - E.g., add two integer, add two doubles

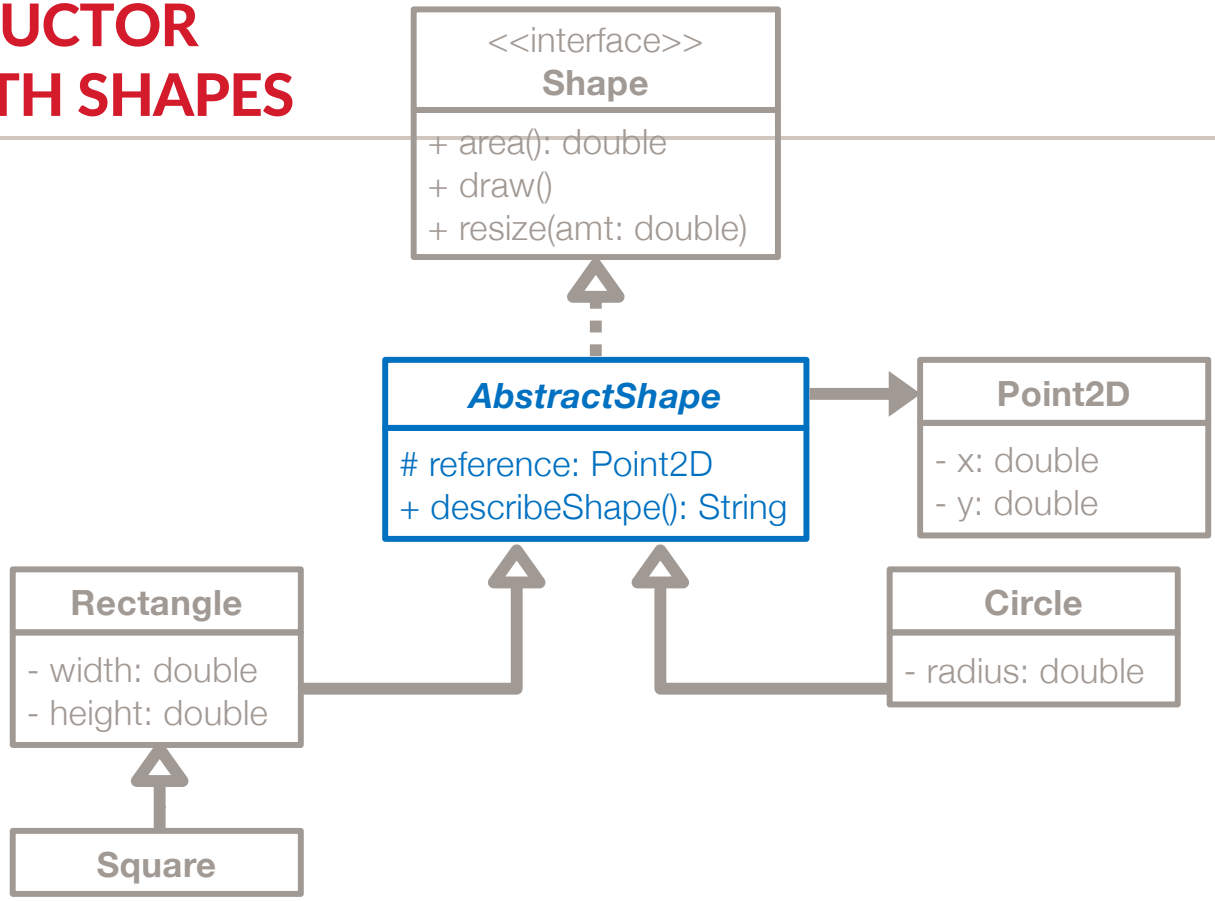
CONSTRUCTOR OVERLOADING

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

EXAMPLE: CONSTRUCTOR OVERLOADING WITH SHAPES

`reference` = the point used to start drawing or resizing.



EXAMPLE: CONSTRUCTOR OVERLOADING

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

- Could we have a “default” reference point?

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

EXAMPLE: OVERLOADING ABSTRACTSHAPE

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

EXAMPLE: OVERLOADING ABSTRACTSHAPE

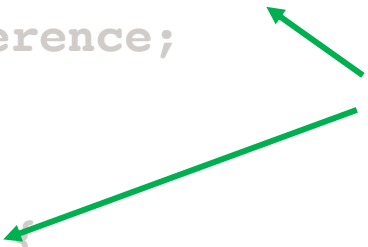
```
public AbstractShape(Point2D reference) {
  this.reference = reference;
}

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

EXAMPLE: OVERLOADING ABSTRACTSHAPE

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

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



Parameters must be different so Java knows which one to use!

EXAMPLE: OVERLOADING ABSTRACTSHAPE

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

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

One parameter, type Point2D

In Circle:

```
public Circle(Point2D reference, double radius) {
  super(reference);
  this.radius = radius;
}
```

One parameter, type Point2D

EXAMPLE: OVERLOADING IN THE CONCRETE CLASSES

```
public Circle(Point2D reference, double radius) {
  super(reference);
  this.radius = radius;
}

public Circle(double radius) {
  super();
  this.radius = radius;
}
```

EXAMPLE: OVERLOADING IN THE CONCRETE CLASSES

```
public Circle(Point2D reference, double radius) {
  super(reference);
  this.radius = radius;
}

public Circle(double radius) {
  super();
  this.radius = radius;
}
```

calls `AbstractShape(Point2D reference)`

calls `AbstractShape()`

EXAMPLE: OVERLOADING IN THE CONCRETE CLASSES

```
public Circle(Point2D reference, double radius) {
  super(reference);
  this.radius = radius;
}

public Circle(double radius) {
  super();
  this.radius = radius;
}
```

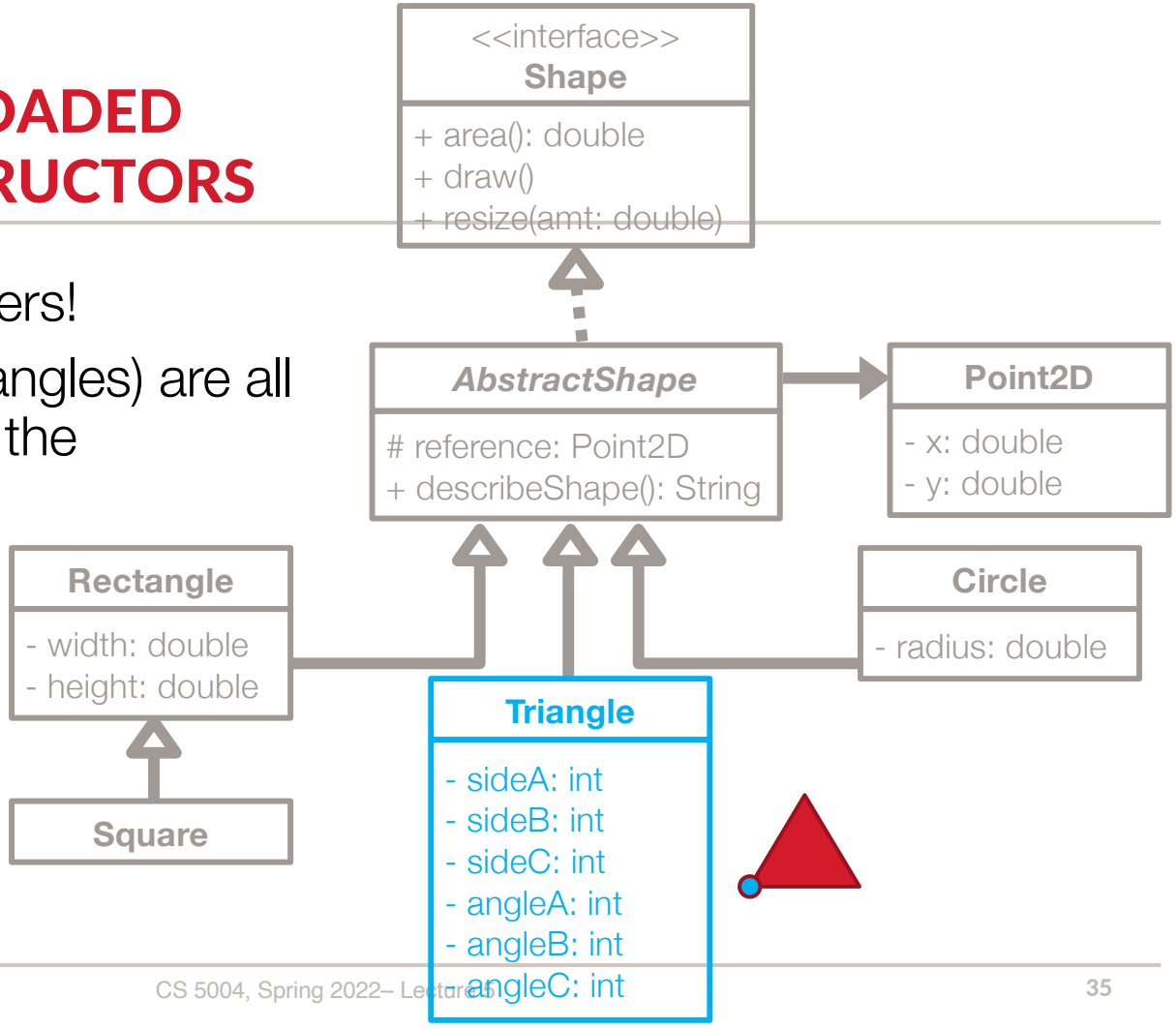
Java automatically calls the appropriate constructor based on arguments passed

```
Circle circle = new Circle(new Point2D(0,0), 5);
Circle circle = new Circle(5);
```

EXERCISE: OVERLOADED TRIANGLE CONSTRUCTORS

That's a lot of parameters!

- Three values (sides, angles) are all we need to calculate the remaining values
  - *One value must be a side*
- What overloaded constructors can we have?
- Don't worry about the math!



THE PROBLEM

```
// One side, two angles
public Triangle(int sideA, int angleA, int angleB) { ...  }

// Two sides, one angle
public Triangle(int sideA, int sideB, int angleA) { ...  }
```

Both constructors have the same signature

```
Triangle triangle1 = new Triangle(30, 30, 30);
```

Which constructor is being called?

WHAT TO DO?

```
// One side, two angles
public Triangle(int sideA, int angleA, int angleB) { ... }

// Two sides, one angle
public Triangle(int sideA, int sideB, int angleA) { ... }
```

Make a design decision

- Determine which option you will support
- Clearly document what the chosen parameters represent

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");
```

```
public class Sum {
    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }
    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }
    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }
    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

JAVA'S LIST INTERFACE

`static <E> List<E>`	`of()`	Returns an unmodifiable list containing zero elements.
`static <E> List<E>`	`of(E e1)`	Returns an unmodifiable list containing one element.
`static <E> List<E>`	`of(E... elements)`	Returns an unmodifiable list containing an arbitrary number of elements.
`static <E> List<E>`	`of(E e1, E e2)`	Returns an unmodifiable list containing two elements.
`static <E> List<E>`	`of(E e1, E e2, E e3)`	Returns an unmodifiable list containing three elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4)`	Returns an unmodifiable list containing four elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5)`	Returns an unmodifiable list containing five elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6)`	Returns an unmodifiable list containing six elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)`	Returns an unmodifiable list containing seven elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)`	Returns an unmodifiable list containing eight elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)`	Returns an unmodifiable list containing nine elements.
`static <E> List<E>`	`of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)`	Returns an unmodifiable list containing ten elements.

DYNAMIC/ RUN-TIME POLYMORPHISM

CS 5004, SPRING 2022– LECTURE 5

SUBTYPE 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.
- Method overriding works because of subtype polymorphism

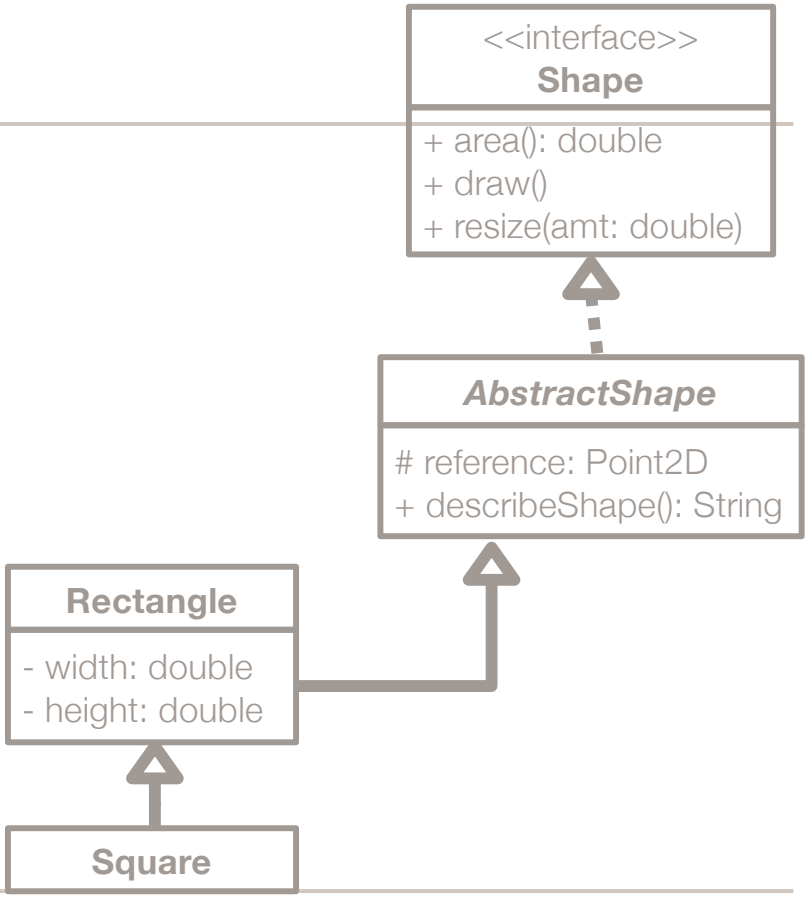
Every object has multiple types

- The type it is instantiated as
- Every type it inherits from
  - Remember: all objects inherit Java's Object class → therefore, every object has *at least* two types

SUBTYPE POLYMORPHISM

What are square's types?

```
Square square = new Square(5);
```

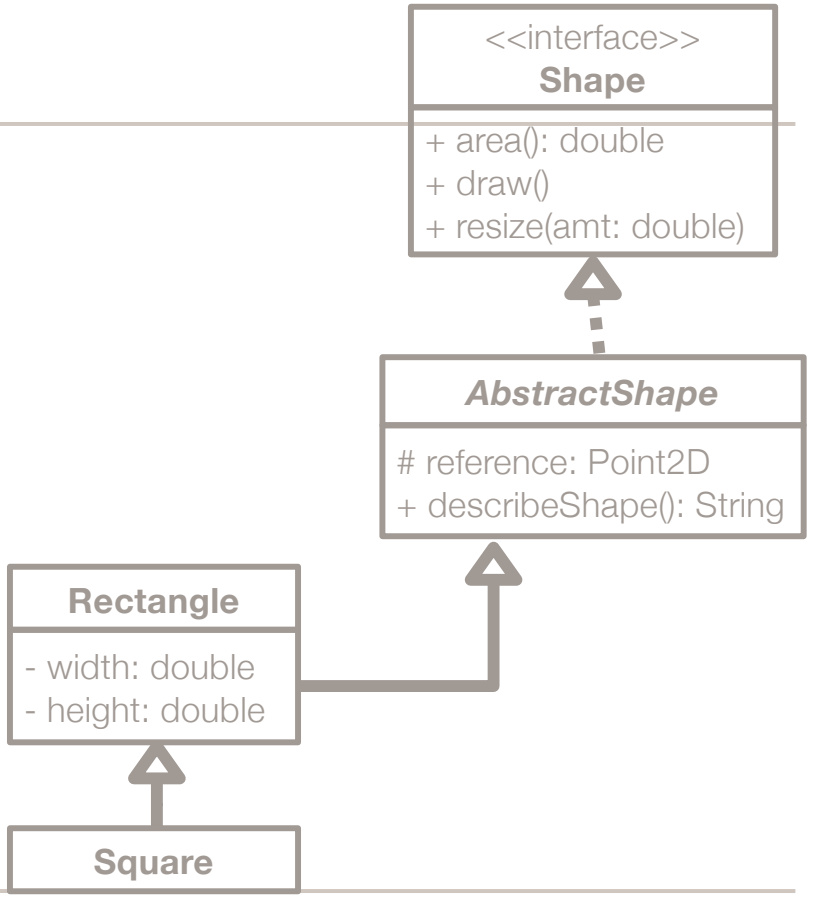


SUBTYPE POLYMORPHISM

What are square's types?

```
Square square = new Square(5);
```

- Square
- Rectangle
- AbstractShape
- Shape
- Object



WE'VE ALREADY BEEN USING SUBTYPE POLYMORPHISM

All **`equals()`** implementations

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
           Objects.equals(getNextNode(), node.getNextNode());
}
```

OBJECTS CAN BE DECLARED AS ANY INHERITED TYPE

Instead of

```
Square square = new Square(5);
```

OBJECTS CAN BE DECLARED AS ANY INHERITED TYPE

Instead of

```
Square square = new Square(5);
```

...we could use

```
Rectangle square = new Square(5);
```

OBJECTS CAN BE DECLARED AS ANY INHERITED TYPE

Instead of

```
Square square = new Square(5);
```

...we could use

```
Rectangle square = new Square(5);
```

...or

```
AbstractShape square = new Square(5);
```

...or

```
Shape square = new Square(5);
```

OBJECTS CAN BE DECLARED AS ANY INHERITED TYPE

Instead of

```
Square square = new Square(5);
```

...we could use

```
Rectangle square = new Square(5);
```

...or

```
AbstractShape square = new Square(5);
```

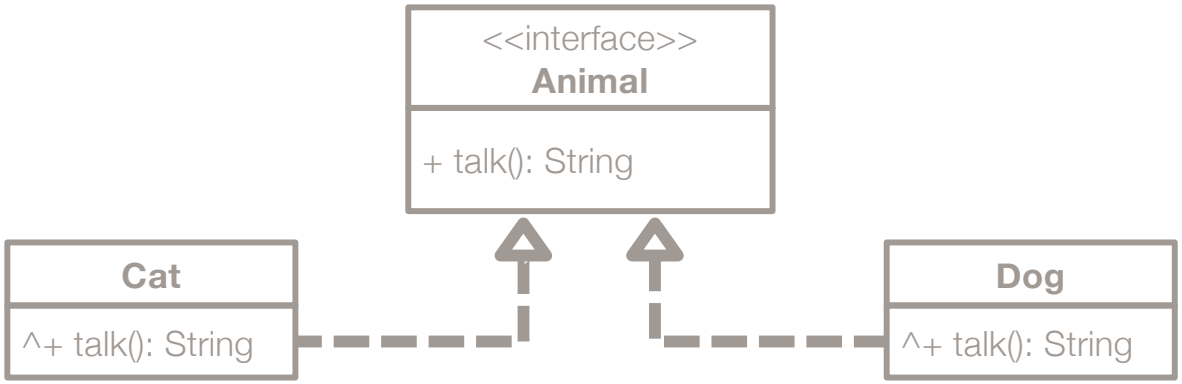
...or

```
Shape square = new Square(5);
```

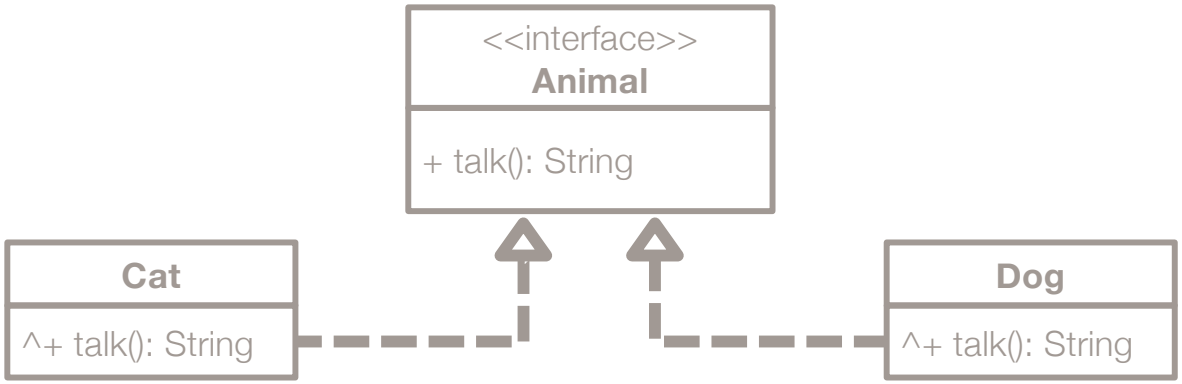
...or even

```
Object square = new Square(5);
```

ANOTHER EXAMPLE OF SUBTYPE POLYMORPHISM

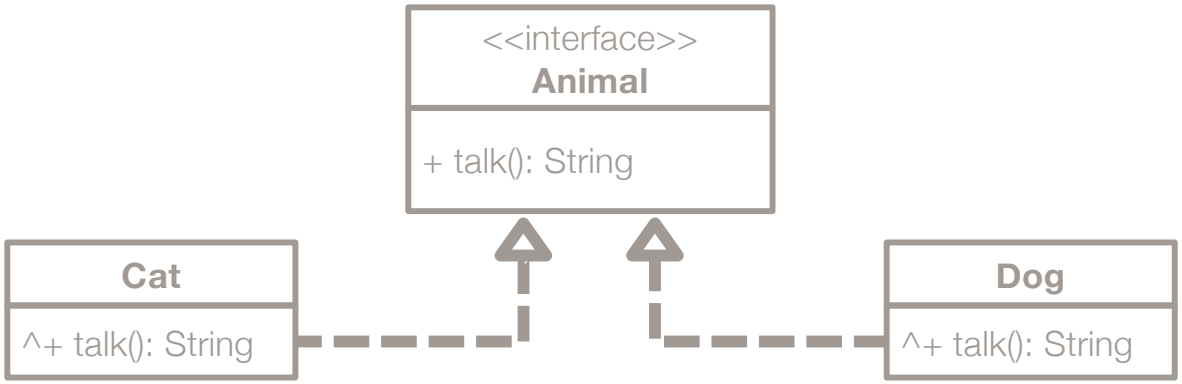


ANOTHER EXAMPLE OF SUBTYPE POLYMORPHISM



- Cat is an Animal
- Dog is an Animal

ANOTHER EXAMPLE OF SUBTYPE POLYMORPHISM



- Cat is an Animal
- Dog is an Animal

Cat and Dog are both **subtypes** of Animal

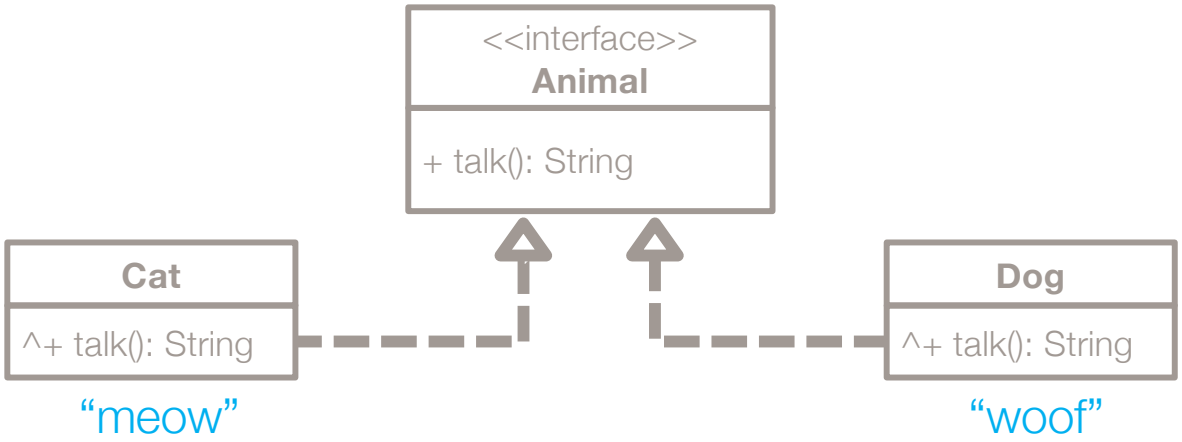
SUBTYPE POLYMORPHISM

Allows us to declare a field or method parameter as "**super type**" and pass it an instance of any "**sub type**"

- Enables a single method/property to work for multiple subtypes

```
public class PetOwner {
  private String name;
  private Animal pet;
  public PetOwner(String name, Animal pet) {
    this.name = name;
    this.pet = pet;
  }
}
```

SUBTYPE POLYMORPHISM

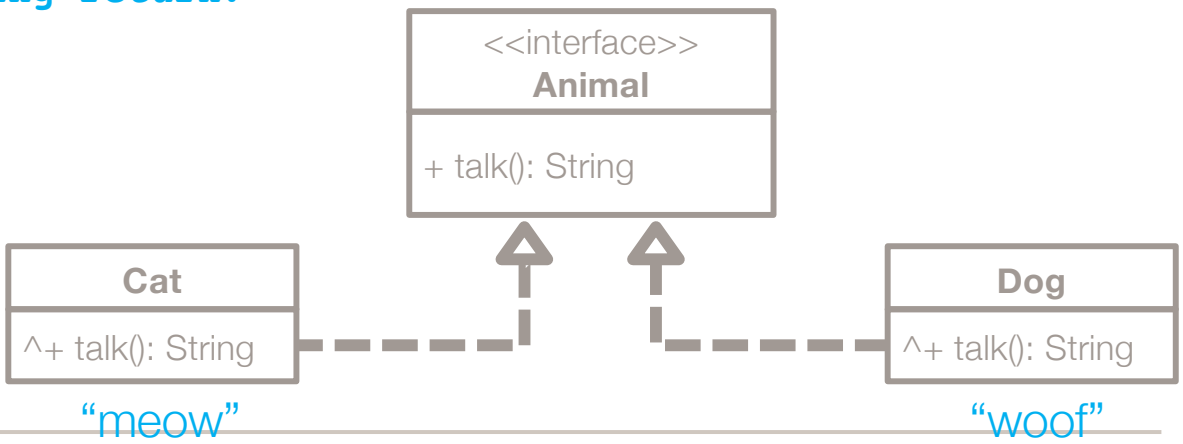


SUBTYPE POLYMORPHISM

```
Animal mittens = new Cat("Mittens");
Animal fido = new Dog("Fido");

PetOwner owner = new PetOwner("Darth Vader", mittens);

// What will the following return?
owner.getPet().talk()
```

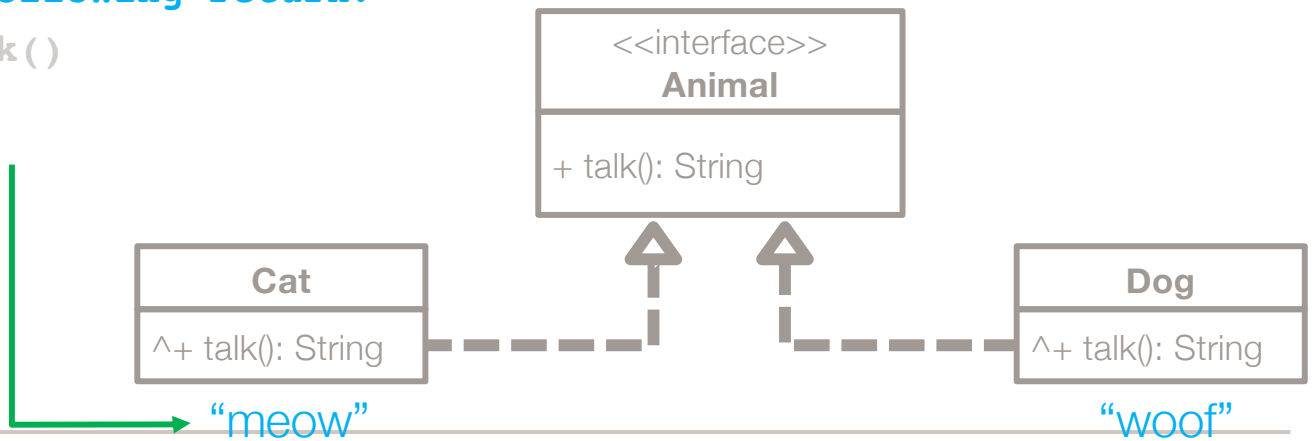


SUBTYPE POLYMORPHISM

```
Animal mittens = new Cat("Mittens");
Animal fido = new Dog("Fido");

PetOwner owner = new PetOwner("Darth Vader", mittens);

// What will the following return?
owner.getPet().talk()
```



COMPILE-TIME VS. RUN-TIME TYPES

With polymorphism, **an object can have a different data type at compile time than at run time.**

```
Animal mittens = new Cat();
Object today = new Date();
```

COMPILE-TIME VS. RUN-TIME TYPES

With polymorphism, **an object can have a different data type at compile time than at run time.**

- Compile-time type = declared type

```
Animal mittens = new Cat();
Object today = new Date();
```

COMPILE-TIME VS. RUN-TIME TYPES

With polymorphism, **an object can have a different data type at compile time than at run time.**

- Compile-time type = declared type
- Runtime type = instantiated type

```
Animal mittens = new Cat();
Object today = new Date();
```

COMPILE-TIME VS. RUN-TIME TYPES

With polymorphism, **an object can have a different data type at compile time than at run time.**

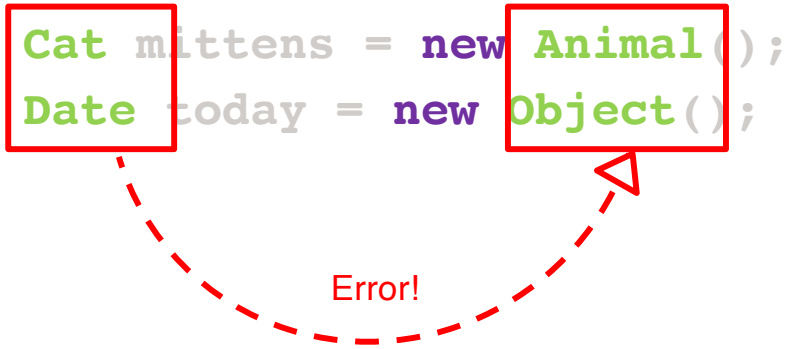
Runtime type MUST be

- the same as compile-time OR
- a subtype of compile-time type

COMPILE-TIME VS. RUN-TIME TYPES

With polymorphism, **an object can have a different data type at compile time than at run time.**

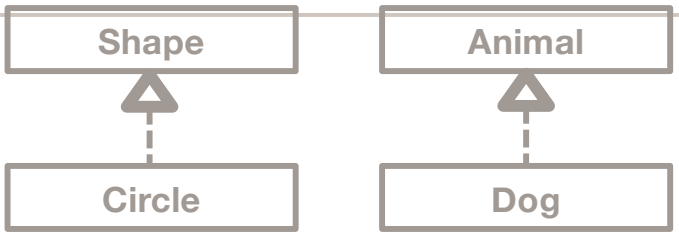
If the types are the wrong way round, you'll get a compile-time error.



SUBTYPING

Goal: Code written to A's specification (Shape, Animal) operates correctly even if given a B.

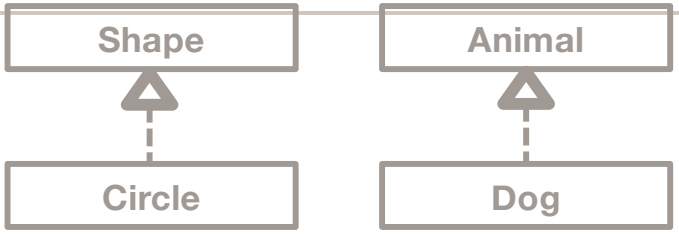
- Clarify design
- Share tests
- (Sometimes) share code



JAVA SUBTYPES VS. "TRUE" SUBTYPES

"B is a subtype of A":

- To be a ***true subtype***, every object that satisfies the rules for a B also satisfies the rules for an A.



SUBTYPING

True subtypes are substitutable for supertypes.

- e.g., an instance of Cat is substitutable for an Animal.
  - If some code expects an Animal and you give it a Cat, it should still work.
- True subtypes meet *all requirements* of the supertype.
- Instances of the subtype don't have more requirements than the supertype.

SUBTYPING

True subtype

B is a true subtype of A if B has a stronger specification than A.

- i.e., B can do all the same things as A (same public methods/fields)
- Easy to achieve when coding to an interface

SUBTYPING

True subtype

B is a true subtype of A if B has a stronger specification than A.

- i.e. B can do all the same things as A (same public methods/fields)
- Easy to achieve when coding to an interface

Java subtype

- e.g., B extends A.
- Not a true subtype if B doesn't meet the same specifications as A
  - Can lead to bugs
  - Can be a bad design practice

SUBTYPING VS. SUBCLASSING

Subtype

- A **specification** notion

Subclass

- An **implementation** notion

SUBTYPING VS. SUBCLASSING

Subtype

- A **specification** notion
- B is a true subtype of A if and only if an object of B can masquerade as an object of A in any context.

Subclass

- An **implementation** notion
- Convenience, factor out repeated code.

SUBTYPING VS. SUBCLASSING

Subtype

- A **specification** notion
- B is a true subtype of A if and only if an object of B can masquerade as an object of A in any context.

Subclass

- An **implementation** notion
- Convenience, factor out repeated code.
- When writing subclasses, aim to make them true subtypes.
  - Not always possible/sensible in practice

SUBSTITUTION PRINCIPLES FOR CLASSES

- If B is a subtype of A, that means B can *always be substituted* for an A.

SUBSTITUTION PRINCIPLES FOR CLASSES

- If B is a subtype of A, that means B can *always be substituted* for an A.
- Any property guaranteed by A must be guaranteed by B.
  - If an instance of the subtype is treated purely as the supertype (only supertype methods/fields used), the result should be consistent with an object of the supertype being manipulated.

SUBSTITUTION PRINCIPLES FOR CLASSES

- If B is a subtype of A, that means B can *always be substituted* for an A.
- Any property guaranteed by A must be guaranteed by B.
- B can *strengthen* a specification
  - Add properties
  - Add/override methods (that match/strengthen the spec)

SUBSTITUTION PRINCIPLES FOR CLASSES

- If B is a subtype of A, that means B can *always be substituted* for an A.
- Any property guaranteed by A must be guaranteed by B.
- B can *strengthen* a specification
- B must not *weaken* a specification
  - Don't remove methods

MATCH/STRENGTHEN/WEAKEN THE SPECIFICATION

Matching the specification of a class

- All non-private fields and methods present in the superclass are present in the subclass (inherited or overridden).

MATCH/STRENGTHEN/WEAKEN THE SPECIFICATION

Matching the specification of a class

- All non-private fields and methods present in the superclass are present in the subclass (inherited or overridden).
- Subclass methods have the same requirements as the superclass version
  - The same number and type of parameters.
  - The same return type.
  - Throw the same or fewer exceptions.

MATCH/STRENGTHEN/WEAKEN THE SPECIFICATION

Strengthening the specification of a class

- Meet all criteria for matching the specification.

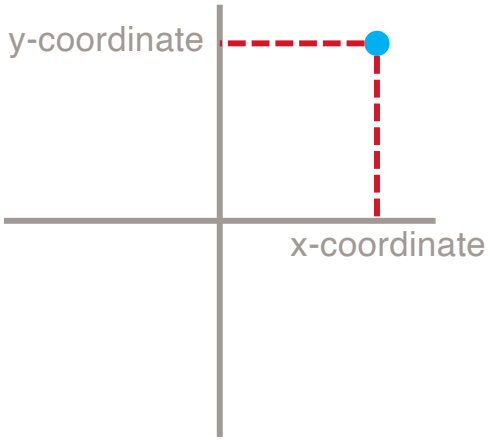
MATCH/STRENGTHEN/WEAKEN THE SPECIFICATION

Strengthening the specification of a class

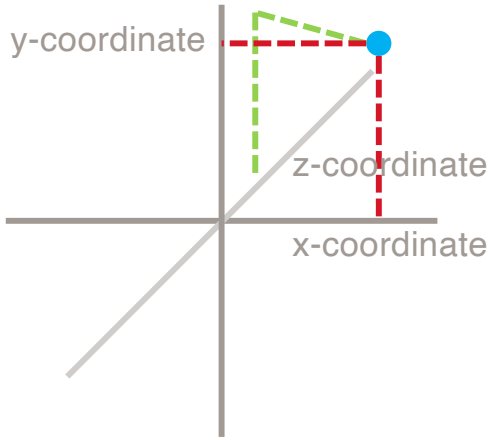
- Meet all criteria for matching the specification.
- Add additional fields/functionality.

EXAMPLE: POINTS IN 2D AND 3D

Point2D = x, y coordinate in 2-dimensional space



Point3D = x, y, z coordinate in 3-dimensional space



POINT3D: TRUE SUBTYPE OF POINT2D

```
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

POINT3D: TRUE SUBTYPE OF POINT2D

```
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

```
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
  public int z() {
    return this.z;
  }
}
```

POINT3D: TRUE SUBTYPE OF POINT2D

```
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

```
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
  public int z() {
    return this.z;
  }
}
```

New field

POINT3D: TRUE SUBTYPE OF POINT2D

```
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

```
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
  public int z() {
    return this.z;
  }
}
```

New method

POINT3D: TRUE SUBTYPE OF POINT2D

```
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int x() {
    return this.x;
  }
  public int y() {
    return this.y;
  }
}
```

Nothing in Point3D will change these

```
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
  public int z() {
    return this.z;
  }
}
```

POINT3D: TRUE SUBTYPE OF POINT2D

```
Point2D point = new Point3D(1, 3, 4);
```

The following Point2D method calls still work as expected for a Point2D type:

```
point.x();
point.y();
```

WHEN A SUBCLASS IS NOT A TRUE SUBTYPE

Superclass: Person

- Name
- Age

WHEN A SUBCLASS IS NOT A TRUE SUBTYPE

Superclass: Person

- Name
- Age

Child extends Person

- Age must be < 18
  - This new requirement means that Child violates the conditions of a true subtype.
    - Does it matter? No, probably not!

IS SQUARE A TRUE SUBTYPE OF RECTANGLE?

Rectangle public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`

Square public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`

IS SQUARE A TRUE SUBTYPE OF RECTANGLE?

Rectangle public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`

Square public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`

YES. Square matches the specification of Rectangle.

IS SQUARE A TRUE SUBTYPE OF RECTANGLE?

Rectangle public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`
- `double resize(double w, double h)`

Square public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`
- `double resize(double w, double h)`
  - Requires w == h
  - Throws InvalidSquareException if not

IS SQUARE A TRUE SUBTYPE OF RECTANGLE?

Rectangle public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`
- `double resize(double w, double h)`

Square public methods:

- `double getWidth()`
- `double getHeight()`
- `double area()`
- `double resize(double amt)`
- `double resize(double w, double h)`
  - Requires w == h
  - Throws InvalidSquareException if not

NO. Square throws an exception where Rectangle would not.

~~Again, not really a problem!~~

STATIC VS. DYNAMIC BINDING

CS 5004, SPRING 2022– LECTURE 5

BINDING

The process of figuring out which implementation of a method is being called.

```
Point2D point
   = new Point3D(1, 3, 4);
point.x();
```

BINDING

The process of figuring out which implementation of a method is being called.

- Java looks for implementation in the class and possibly parent classes.

```
Point2D point
   = new Point3D(1, 3, 4);
point.x();
```

BINDING

The process of figuring out which implementation of a method is being called.

- Java looks for implementation in the class and possibly parent classes.

```
Point2D point
   = new Point3D(1, 3, 4);
point.x();
```

Look for a method, `x()`, in Point3D. If not found, check Point2D

BINDING

Can happen at two different times:

- Compile time
- Run time

BINDING

Can happen at two different times:

- Compile time = **static binding**
- Run time = **dynamic binding**

STATIC BINDING

Done by the compiler.

- If an object's runtime type can be determined, its method calls will be bound statically.

```
Cat mittens
   = new Cat("Mittens");
mittens.talk();
```

STATIC BINDING

Done by the compiler.

- If an object's runtime type can be determined, its method calls will be bound statically.

```
Cat mittens
   = new Cat("Mittens");
mittens.talk();
```

Compiler knows the runtime type of `mittens` (Cat). `mittens.talk()` will be bound to the implementation of `talk()` in the Cat class.

DYNAMIC BINDING

Happens at runtime.

- When the compiler can't tell exactly what the runtime type will be

From the PetOwner class:

```
private Animal pet;
…
this.pet.talk();
```

DYNAMIC BINDING

Happens at runtime.

- When the compiler can't tell exactly what the runtime type will be

From the PetOwner class:

```
private Animal pet;
…
this.pet.talk();
```

Compiler doesn't know which subclass of Animal's implementation to use.
JVM binds it at runtime.

STATIC VS DYNAMIC TYPES

- static type == compile time type
- dynamic type == runtime type

```
Animal mittens = new Cat();
Object today = new Date();
```

STATIC VS DYNAMIC TYPES

- **static type==compile time type**
- dynamic type == runtime type

```
Animal mittens = new Cat();
Object today = new Date();
```

STATIC VS DYNAMIC TYPES

- static type == compile time type
- **dynamic type == runtime type**

```
Animal mittens = new Cat();
Object today = new Date();
```

CAUTION

When an object has different compile-time and run-time types, only the compile-time type's properties/methods are accessible

```
Point3D aPoint = new Point3D(1, 2, 4);
Point2D anotherPoint = new Point3D(1, 2, 4);
```

`aPoint.getZ();` → compiles

`anotherPoint.getZ();` → doesn't compile, **`getZ`** does not exist in compile-time type

CHECKING OBJECT TYPE AND CASTING

CS 5004, SPRING 2022– LECTURE 5

CHECKING AN OBJECT'S TYPE

Sometimes it's useful to check an object's type at runtime...

e.g., is `this.pet` actually a Dog or a Cat?

```
private Animal pet;
…
this.pet.talk();
```

THE `INSTANCEOF` OPERATOR

Tests whether a given object is a given type:

- `variable instanceof SomeType`

```
if (species instanceof Cat)
{
  // do something
}
```

THE `INSTANCEOF` OPERATOR: CAUTION!

`instanceof` returns true for multiple types!

- All objects implicitly extend Java's `Object` base class.
- Objects that inherit other classes will be `instanceof` those classes too.
- If you must branch code based on type, make sure to check *most specific* types first.

INSTANCEOF RETURNS TRUE FOR MULTIPLE TYPES

```
PetOwner owner = new PetOwner("Darth Vader",
                 new Cat("Mittens"));
```

In the PetOwner constructor:

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}
```

`INSTANCEOF` RETURNS TRUE FOR MULTIPLE TYPES

```
PetOwner owner = new PetOwner("Darth Vader",
                              new Cat("Mittens"));
```

In the PetOwner constructor:

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}
```

The `owner` object's `pet` field has runtime type of `Cat`.

`INSTANCEOF` RETURNS TRUE FOR MULTIPLE TYPES

All of the following will return true.

```
owner.pet instanceof Cat
owner.pet instanceof Animal
owner.pet instanceof Object
```

CASTING FROM ONE DATA TYPE TO ANOTHER

If we know what type an object is, we can "cast" it from one data type to another:

- if both types are in the same inheritance tree
- …and the object already possesses the type you want to cast to

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Currently, `point` can't access functionality specific to the Point3D runtime datatype. E.g. the `z()` method.

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Currently, `point` can't access functionality specific to the Point3D runtime datatype. E.g. the `z()` method.

Because `point` is *already an instance of* `Point3D`, we can cast it to `Point3D` in order to access the functionality.

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Two options:

- Temporarily cast “in place”.
- Create a new object and cast a copy of the current object.

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Two options:

- **Temporarily cast "in place".**

  ```
  ((Point3D) point).z();
  ```

- Create a new object and cast a copy of the current object.

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Two options:

- Temporarily cast “in place”.

```
((Point3D) point).z();
```

- **Create a new object and cast a copy of the current object.**

```
Point3D newPoint = (Point3D) point;
```

CASTING: POINT2D AND POINT3D

```
Point2D point = new Point3D();
```

Two options:

- Temporarily cast "in place"

```
((Point3D) point).z();
```

- Create a new object and cast a copy of the current object.

```
Point3D newPoint = (Point3D) point;
```

Syntax:
Put the data type you want to cast to in ( ) in front of the variable.

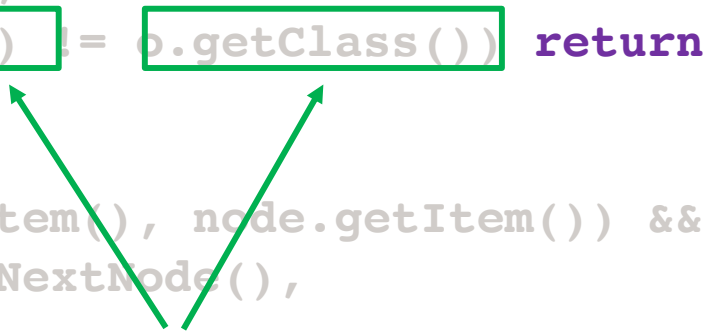
IMPORTANT: CHECK BEFORE CASTING

An object can only be cast from one data type to another if it is an instance of that type. Always check with **`instanceof`**!

```
Point2D point = new Point3D();
if (point instanceof Point3D) {
  Point3D newPoint = (Point3D)point;
}
```

ONE MORE WAY TO CHECK TYPE: GETCLASS

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
           Objects.equals(getNextNode(),
node.getNextNode());
}
```



The runtime type of the object

- Can only be one class

GOOD OOD PRACTICE

CS 5004, SPRING 2022– LECTURE 5

GOOD OOD PRACTICE

- Don’t use instanceof/getClass when you can take advantage of polymorphism
- Parent classes should know nothing about their children

NOW THAT YOU KNOW ABOUT `INSTANCEOF` AND `GETCLASS`...

...avoid use outside the equals method!

- Use polymorphism (overriding) instead

```
public class PetOwner {
  private String name;
  private Animal pet;
  public PetOwner(String name, Animal pet) { ...}

  public void feedPet() { ... }
}
```

DODGY DESIGN

```
public void feedPet() {
  if (this.pet instanceof Cat) {
    // Do Cat-specific thing
  }
  else if (this.pet instanceof Dog) [
    // Do Dog-specific thing
  }
}
```

Not extensible – *what if we add more types of pet?*
Breaks encapsulation – *species-specific behavior should be in the species' class*

THE OOD APPROACH

In Animal:

```
void eat(); // Implement in each subclass
```

In PetOwner:

```
public void feedPet() {
  this.pet.eat();
}
```

DODGY DESIGN

Parents should not need knowledge of their children!

In Parent:

```
String foo(Child1 child) {...}
String foo(Child2 child) {...}
String foo(Child3 child) {...}
```

Not extensible – *what if we add more child classes?*
Breaks encapsulation – *child-specific behavior should be in the child class*

```
String foo() {
  if (this instanceof Child1)
    return "A";
  else if (this instanceof Child2)
    return "B";
  else if (this instanceof Child3)
    return "C";
}
```

SUMMARY

CS 5004, SPRING 2022– LECTURE 5

COMPILE TIME AND RUN-TIME

- Java programs have two distinct phases in their lifetimes:
  - Compile time (static time) - refers to source code, and the point in time when the source code is being compiled by the Java compiler (think of a compiler as a translator)
  - Run time (dynamic time) – refers to when the code is being evaluated (or executed or run)by the Java Virtual Machine (JVM)

COMPILE TIME (STATIC) AND RUN-TIME (DYNAMIC) DATA TYPE

```
Person emily = new Person();
Singer adele = new Singer();
Person flora = new Singer();
```

- Static (compile time) type – the declared type of a reference variable. Used by a compiler to check syntax
- Dynamic (run time) type – the type of an object that the reference variable currently refers to (it can change as the program execution progresses)

STATIC AND DYNAMIC TYPES

- Binding – the process of bounding a method call (method invocation) to one of its implementations
  - Involves method lookup in the class, or one or its parents
  - Both method names and parameters are checked

- Binding can happen at two different times
  - Compile time == static binding
  - Run time == dynamic binding

STATIC BINDING

- References have a type
  - (they refer to instances of a particular Java class)
- Objects have a type
  - Instances of a particular Java class
  - Instances of all of their super-class
- Static binding done by the compiler (when it can determine the type of an object)
- Method calls are bound to their implementation during the compilation

DYNAMIC BINDING

- Achieved at runtime
  - Data type of an object cannot be determined at compile time
  - JVM (not the compiler) binds a method call to its implementation

- Instances of a sub-class can be treated as if they were an instance of the parent class
  - Therefore the compiler doesn't know its type, just its base type

DYNAMIC BINDING

- Whenever a reference refers to an interface or a base class, methods are dynamically bound
  - Method implementation determined at runtime

- Polymorphism and dynamic binding are inter-connected, and represent a powerful feature of OO design

- Allow the creation of "frameworks"
  - Applications that are implemented around interfaces, but are customised by plugging in different implementations of those interfaces
  - Very extensible

CASTING

- Casting - a Java language feature that allows us to alter the compile-time type of a variable
- The runtime type is not altered because of a cast
- We can explicitly cast to a compile-time type using (T) o
- We can also implicitly cast using subtype polymorphism

- Types of casts:
- Upcasting - when we cast from a subclass to a superclass (or interface) (since we are moving up in the class hierarchy)
- Downcasting – when we cast from a superclass (or interface) to a subclass (every time we are moving down the class hierarchy)
  - Down casts are dangerous
  - We have to write code to ensure that our down cast is safe

CHECKING AN OBJECT TYPE

- It is possible to check the run time type of an object
  - When: if we only have a reference to an interface or base class
- Use the `instanceof` operator
  - Must be applied to an object, tests whether it has a given type

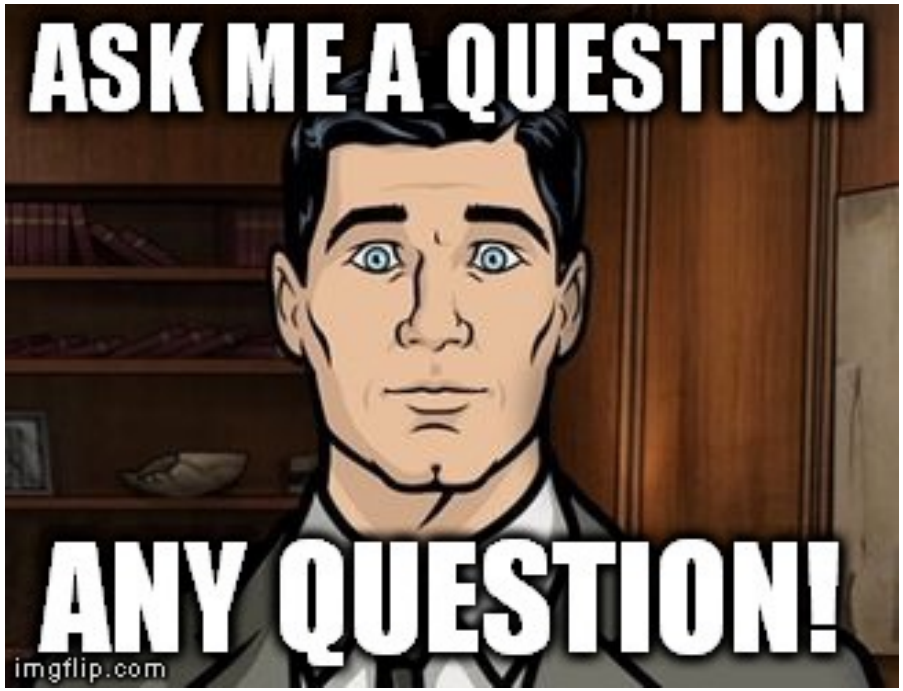
CASTING

- If we know the type, we can then “cast” the object

```
Vehicle bike = new MotorBike();
if (bike instanceof MotorBike) {
  MotoBike bike = (MotorBike)bike;
}
```

- If the object is not of the cased that type, an exception will be thrown
  - Good idea to always check before casting, unless you’re absolutely sure!

YOUR QUESTIONS



[Meme credit: imgflip.com]

REFERENCES AND READING MATERIAL

- Java Getting Started (https://docs.oracle.com/javase/tutorial/getStarted/index.html)
- Object-Oriented Programming Concepts (https://docs.oracle.com/javase/tutorial/java/concepts/index.html)
- Language Basics (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)
- How to Design Classes (HtDC), Chapters 1-3
- JUnit: Getting Started (https://github.com/junit-team/junit4/wiki/Getting-started)
- JUnit: Assertions (https://github.com/junit-team/junit4/wiki/Assertions)
- Unit testing with JUnit: http:/www.vogella.com/tutorials/JUnit/article.html
- Java Tutorial: Interfaces and Inheritance: https://docs.oracle.com/javase/tutorial/java/IandI/index.html
- Java – Exceptions (https://www.tutorialspoint.com/java/java_exceptions.htm)
- Declare Your Own Exception (https://www.ibm.com/developerworks/community/blogs/738b7897-cd38-4f24-9f05-48dd69116837/entry/declare_your_own_java_exceptions?lang=en)
- Geeks for Geeks: Arrays in Java: https://www.geeksforgeeks.org/arrays-in-java/