CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2024

LECTURE 3

Tamara Bonaci
t.bonaci@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

ADMINISTRIVIA

AGENDA

- Review
  - Classes and objects
  - Immutability
- Exceptions and unit testing
- Inheritance – part 1
  - Everything is an object
  - Equality (methods `equals`() and `hashCode`())
- Inheritance – part 2
  - Interfaces and abstract classes
- Enumerations and the switch statement
- Good OOD practice

REVIEW

CS 5004, SPRING 2024 – LECTURE 3

REVIEW: OBJECTS AND CLASSES

- Object – an entity consisting of states and behavior
  - States stored in variables/fields
  - Behavior represented through methods
- Class – template/blueprint describing the states and the behavior that an object of that type supports
- Classes consist of:
  - Local variables – variables defined within any method, constructor or block
  - Instance variables – variables within a class, but outside any method
  - Class variables – variables declared within a class, outside of any method, with the keyword static

REVIEW: IMMUTABLE OBJECTS

- Object whose internal state remains constant after it has been entirely created
- Strategies for defining immutable objects:
  - Make all fields `private` and `final`
  - Don't provide setter methods
  - Don't allow subclasses to override methods
  - If the instance fields include references to mutable objects, don't allow those objects to be changed

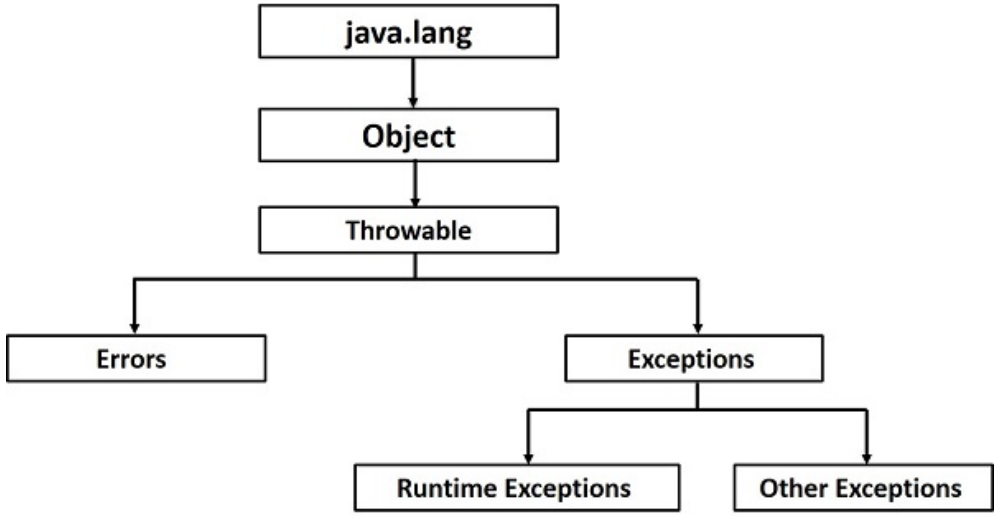
EXCEPTIONS

CS 5004, SPRING 2024 – LECTURE 3

EXCEPTIONS

- Exception occurs when something unexpected happens during the program execution

- All exception classes in Java are subtypes of the java.lang.Exception class



[Figure credit:https://www.tutorialspoint.com/java/java_exceptions.htm]

WRITING A METHOD WITH AN EXCEPTION

```
public void deposit(Double amount) throws IncorrectDepositAmountException {

  if(amount > 0) {
    if (amount <= DEPOSIT_MAX) {
      this.balance += amount;
    }else {
      throw new IncorrectDepositAmountException("Your deposit amount exceeds the daily limit!");
    }
  }
  else{
    throw new IncorrectDepositAmountException("You cannot deposit negative amount!");
  }
}
```

In method signature, declare that an exception can be thrown

Throw an exception

CALLING A METHOD WITH AN EXCEPTION

```
public static void main(String [] args) throws IncorrectDepositAmountException {
    CheckingAccount checkingAccount = new CheckingAccount( number: 101);

    System.out.println("Depositing $500...");
    checkingAccount.deposit( amount: 500.00);

    try {
        System.out.println("\n Withdrawing $100...");
        checkingAccount.withdraw( amount: 100.00);

        System.out.println("\nWithdrawing $600...");
        checkingAccount.withdraw( amount: 600.00);
    }catch(InsufficientFundsException | IncorrectWithdrawalAmountException e) {
        System.out.println("Sorry, but you are short $");
    }
    try {
        checkingAccount.withdraw( amount: -700.00);
    } catch (InsufficientFundsException | IncorrectWithdrawalAmountException e) {
        e.printStackTrace();
    }
}
}
```

Try-catch block:
Start by executing `try` block
If either exception `InsufficientFundsException` or `IncorrectWithdrawalExceeption` is caught, execute code in the `catch` block

TESTING A METHOD WITH AN EXCEPTION

- When testing a method that can throw an exception, there are several things to check:
  - That the method is functionally correct within its valid operating range (happy path)
  - That the method does not throw an exception when it is not supposed to (if it does, a test fails)
  - That the method throws an exception when expected (if it doesn't, a test fails)

TESTING A METHOD WITH AN EXCEPTION – JUNIT 5

- Writing a test that passes when an expected exception is thrown:

```
@Test
void testExpectedException() {

  //First argument - specifies the expected exception.
  //Here it expects that code block will throw NumberFormatException
  //Second argument - is used to pass an executable code block or lambda expression
  Assertions.assertThrows(NumberFormatException.class, () -> {
    Integer.parseInt("One");
  });

}
```

[Image credit:https://howtodoinjava.com/junit5/expected-exception-example/}

TESTING A METHOD WITH EXCEPTION

```
class CheckingAccountTest {

    private CheckingAccount testAccount;

    @BeforeEach
    void setUp() { testAccount = new CheckingAccount( number: 102); }

    @Test
    void deposit() throws IllegalDepositAmountException {
        Double expectedValue = 200.0;
        testAccount.deposit(expectedValue);
        assertEquals( expected: 200.0, testAccount.getBalance(),  delta: 0.001);
    }
```

TESTING A METHOD WITH EXCEPTION

Writing a test that passes when an expected exception is thrown:

```
@Test
void depositException() {
    assertThrows(IllegalDepositAmountException.class, () -> {testAccount.deposit( amount: -300.0);});
}
```

WRITING YOUR OWN EXCEPTIONS

- You can create your own exceptions in Java

- Some rules:
  - All exceptions inherit the behavior of class `Throwable`
  - If you want to write a checked exception, extend class `Exception`
  - If you want to write a runtime exception, extend class `RuntimeException`

EXAMPLE: CUSTOM INSSUFFICIENTFUNDSEXCEPTION

```java
package exceptions.bankingSystem;

public class InsufficientFundsException extends Exception {
  private Double amount;

  public InsufficientFundsException(Double amount) {

    this.amount = amount;
  }

  public Double getAmount() { return amount; }
}
```

INHERITANCE – PART 1

CS 5004, SPRING 2024 – LECTURE 3

INHERITANCE AND "IS A" RELATIONSHIP

- Inheritance - set of classes connected by an 'is-a' relationships
- 'Is-a' relationship - hierarchical connection where one category can be treated as a specialized version of another
  - Example 1:
    - Every student is a person
    - Every ALIGN student is a student
  - Example 2:
    - Every pepper is a vegetable
    - Every bell pepper is a pepper
    - Every banana pepper is a pepper

CLASS INHERITANCE

- Many programming languages (Java, C++, C#) provide a direct support for is-a relationship through class inheritance

- Class inheritance - new class extends existing class
  - Original/Extended class (also known as base class or super class)
  - New/Extending class (also known as derived class or subclass)

- Rules for derived classes (subclasses):
  - Derived class automatically inherits all NON-private instance variables and methods of the base class
  - Derived class can add additional methods and instance variables
  - Derived class can provide different versions of inherited methods

- Note: in Java, a class can extend only one class

EVERYTHING IS AN OBJECT IN JAVA

- `public class Object` – the root of the class hierarchy
  - Every class has `Object` as a superclass
  - All objects inherit public methods of `Object`

`protected Object clone()`	Creates and returns a copy of this object.
`Boolean equals (Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void finalize()`	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?> getClass()`	Returns the runtime class of this Object.
`int hashCode()`	Returns a hash code value for the object.
`void notify()`	Wakes up a single thread that is waiting on this object's monitor.
`String toString()`	Returns a string representation of the object.

EXPECTED PROPERTIES OF EQUALITY

- Reflexive: `a.equals(a) == true`
  - Confusing if an object does not equal itself
- Symmetric: `a.equals(b)` ←→ `b.equals(a)`
  - Confusing if order-of-arguments matters
- Transitive: `a.equals(b) && b.equals(c)` → `a.equals(c)`
  - Confusing again to violate centuries of logical reasoning
    - A relation that is reflexive, transitive, and symmetric is an *equivalence relation*

SPECIFICATION FOR METHOD EQUALS()

- Method **`Boolean equals (Object o)`** - indicates whether some object is “equal to” another object one

- Needs to satisfy the following properties:
  - Reflexive: `x.equals(x)` → true
  - Symmetric: `x.equals(y)` iff `y.equals(x)`
  - Transitive: `x.equals(y)` and `y.equals(z)` then `x.equals(z)`
  - Consistent: `x.equals(y)` Should always return the same value (assuming neither object is modified)
  - If x is not null, `x.equals(null)` → false

IMPLEMENTING METHOD EQUALS()

- Use `@Override` notation
- Overridden method must have signature:

```
public Boolean equals( Object o)
```

- You choose how to determine equality, but there are some basic steps:
  1. Test this == o →return true without further checking
  2. Test o instanceof <current class> → return false if not
  3. Compare fields as appropriate

METHOD HASHCODE()

- Another method in `Object`- `public int hashCode()`
- Computes a unique(ish) integer key (a hash) from an object, for compatibility with hashing data structures

- Contract (again essential for correct overriding):
  - Self-consistent: `o.hashCode() == o.hashCode()`
- as long as o doesn't change between the calls
  - Consistent with equality:
- `a.equals(b)` → `a.hashCode() == b.hashCode()`

TESTING METHODS EQUALS() AND HASHCODE()

- Yes, you should test these methods!

- Your test should check that methods correctly satisfy their contracts
- You may want to break down the test method into several simpler test methods, each testing one requirement from the contract

INHERITANCE – PART 2

CS 5004, SPRING 2024 – LECTURE 3

COMPOSITION

- Composition - set of classes connected by an ‘has-a’ relationships
- ‘Has-a’ relationship – a relationship where one class can use the functionality of another class by using an instance of that class
- Example 1:
  - Every person has a name
  - Every person has a date of birth
- Example 2:
  - Every vehicle has a make
  - Every vehicle has a model
  - Every vehicle has a manufacturing year

COMPOSITION VS. INHERITANCE

- Composition - set of classes connected by an ‘has-a’ relationships
- Example:
  - Every person has a name
  - Every person has a date of birth

- Inheritance - set of classes connected by an ‘is-a’ relationships
- ‘Is-a’ relationship - hierarchical connection where one category can be treated as a specialized version of another
  - Example:
    - Every student is a person
    - Every ALIGN student is a student

OTHER TYPES OF INHERITANCE

- Interfaces - provide a template but no implementation
- Abstract classes - provides some implementation, but not all

WHAT IS AN INTERFACE?

- A set of ***method declarations***—a template for what a class can do.

```
public interface MyInterface {

  void requiredMethod1();

  boolean requiredMethod2(int param);

}
```

WHAT IS AN INTERFACE?

A set of ***method declarations***—a template for what a class can do

- Cannot be *instantiated* – no constructor
- Does not actually implement the methods it declares
- All methods are **public** by default
- Can contain only static fields

WHAT IS AN INTERFACE?

Classes can ***implement*** interfaces

- Classes fill in the implementation details of methods declared in an interface
- One class can implement multiple interfaces
  - …but *extend* only one super class.

```
public class MyClass implements MyInterface {

  void requiredMethod1() {
    // Do something
  }

  boolean requiredMethod2(int param) {
    return param == 0;
  }
}
```

WHEN IS AN INTERFACE USEFUL?

- Whenever you can imagine a “category” of classes that must have some common behavior
- AND implementation of common behavior needs to look different for each some/each of the classes

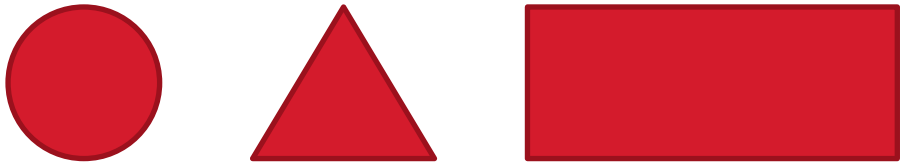
WHEN IS AN INTERFACE USEFUL?

- Whenever you can imagine a “category” of classes that must have some common behavior
- AND implementation of common behavior needs to look different for each some/each of the classes

- Example: **what do the following have in common?**

WHEN IS AN INTERFACE USEFUL?

What do the following have in common?



Example: A Shape interface

- `area()` – gets the area of a shape
- `draw()` – draws a shape
- `resize(double amt)` – resizes a shape by `amt`

WHEN IS AN INTERFACE USEFUL?

Example: A Shape interface

- `area()` – gets the area of a shape
- `draw()` – draws a shape
- `resize(double amt)` – resizes a shape by `amt`

All shapes should support those methods
BUT implementation will be very different

BASIC INTERFACE STRUCTURE

Interfaces are created in their own files (like a class).

```
public interface Shape {
  // An empty interface called "Shape"
}
```

BASIC INTERFACE STRUCTURE

Interfaces are created in their own files (like a class, or an enum)

Note the keyword, **`interface`**

```
public interface Shape {
   // An empty interface called "Shape"
}
```

BASIC INTERFACE STRUCTURE

Interfaces contain only method **signatures**, with the format:

```
<return type> methodName(<type and name of any parameters>);
```

```
public interface Shape {
   void area();
   void draw();
   double resize(double amt);
}
```

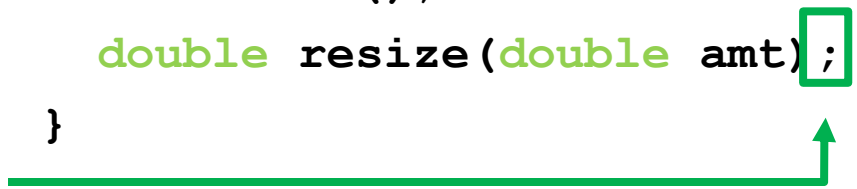
BASIC INTERFACE STRUCTURE

Interfaces contain only method **signatures**, with the format:

```
<return type> methodName(<type and name of any parameters>);
```

Note the semicolon and lack of curly braces after each declaration!

```
public interface Shape {
   void area();
   void draw();
   double resize(double amt);
}
```



IMPLEMENTING AN INTERFACE IN A CLASS

```
class Rectangle implements Shape {

}
```

IMPLEMENTING AN INTERFACE IN A CLASS

```
class Rectangle implements Shape {

}
```

Indicates that this is an implementation of an interface

IMPLEMENTING AN INTERFACE IN A CLASS

```
class Rectangle implements Shape {

}
```

The interface to be implemented

ABSTRACTING OUT COMMON BEHAVIOR

- Example – shapes
  - There exists different types of shapes: Circle, Square, Rectangle
  - Circle has a pin (its center) and a radius
  - Square has a pin (the top left corner) and a side
  - Rectangle has a pin (the top left corner), a width and a height

- Question: is there anything common for all the shapes above?
- Answer: Yes, a pin
- What should we do about it?
- Abstract it in a common class, Shape

Does it make sense to be able to instantiate object Shape?

CONCRETE VS. ABSTRACT CLASSES

Concrete classes

- *Every class you've written so far.*

Abstract classes

CONCRETE VS. ABSTRACT CLASSES

Concrete classes

- **Fully** implemented
  - constructor, all methods implemented.

Abstract classes

- **Partially** implemented
  - may contain “abstract” methods.
  - can also contain implemented methods.

CONCRETE VS. ABSTRACT CLASSES

Concrete classes

- **Fully** implemented
- If implementing an interface, **must** implement all interface methods!

Abstract classes

- **Partially** implemented
- If implementing an interface, **don't have to** implement all interface methods.

CONCRETE VS. ABSTRACT CLASSES

Concrete classes

- **Fully** implemented
- If implementing an interface, **must** implement all interface methods!
- Instantiated directly

Abstract classes

- **Partially** implemented
- If implementing an interface, **don't have to** implement all interface methods.
- Can't be instantiated directly.

WHEN TO USE AN ABSTRACT CLASS?

Instead of (or as well as) as an interface:

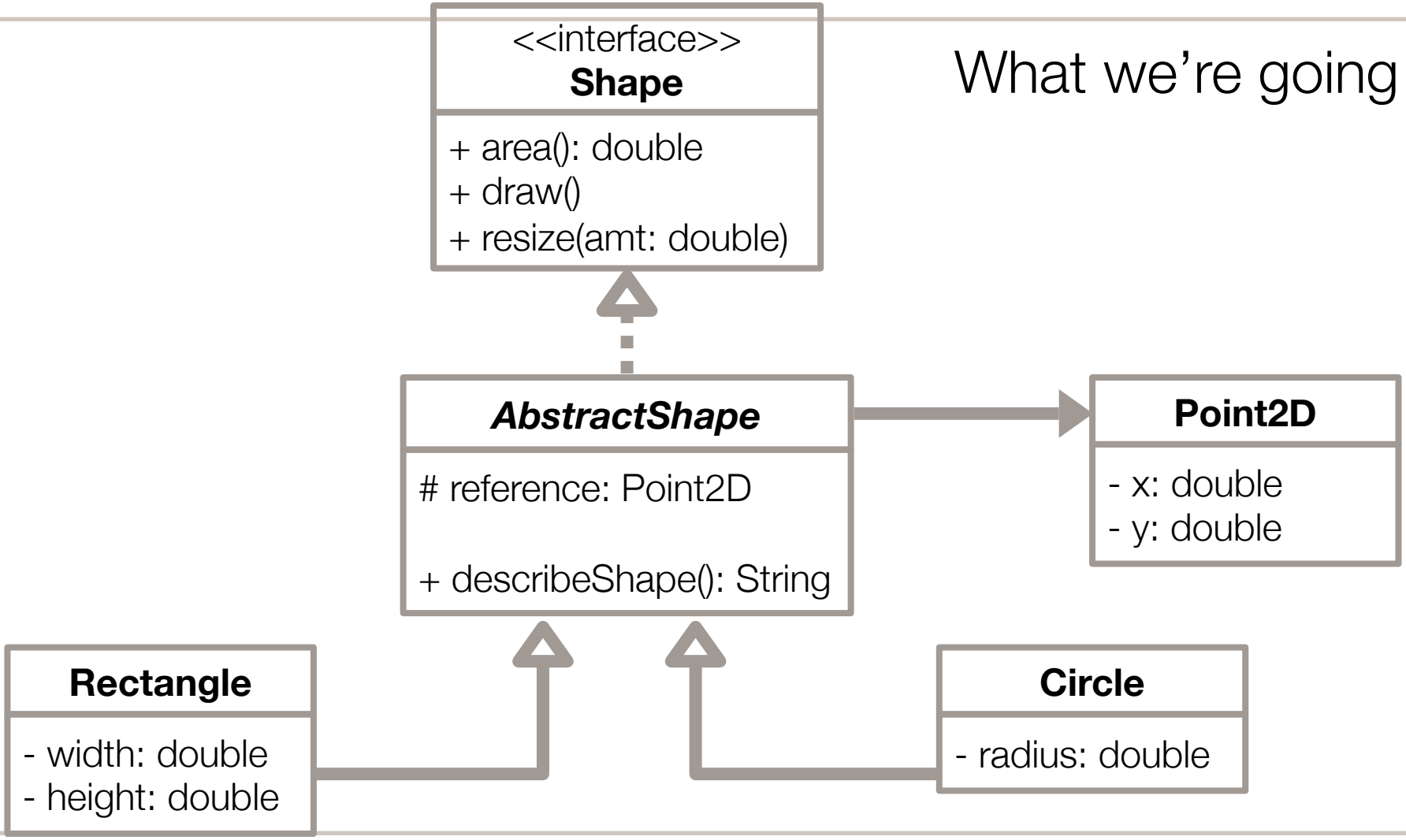
- When you want to provide *some* implementation details common to multiple potential subclasses.

Instead of a concrete class:

- When you don't want users to instantiate the class directly.

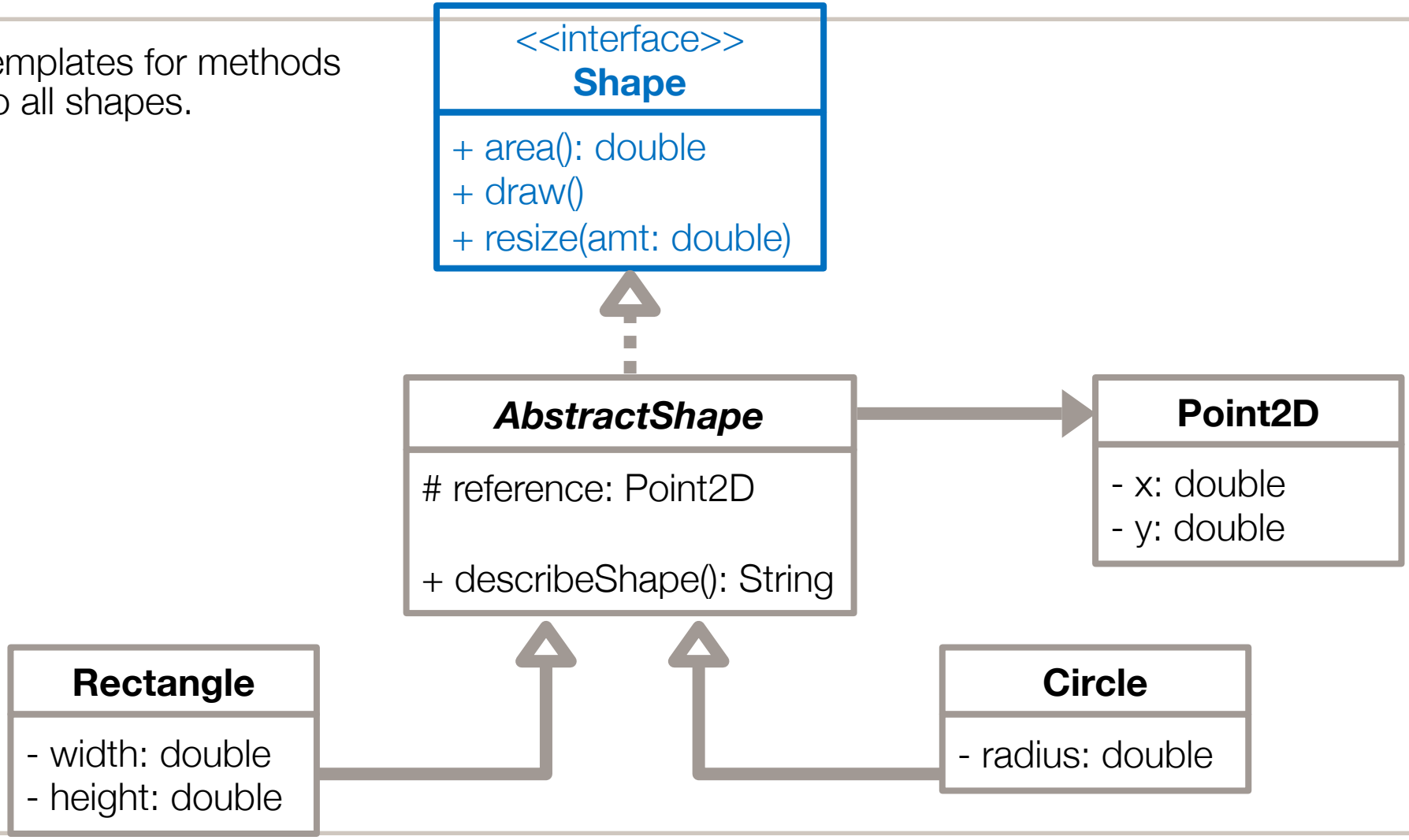
EXAMPLE - MORE ON SHAPES

What we're going to build



EXAMPLE - MORE ON SHAPES

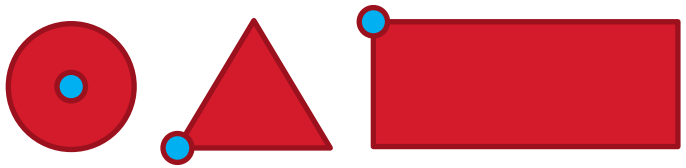
Provides templates for methods common to all shapes.



EXAMPLE - MORE ON SHAPES

Abstract class, implements Shape.

Initializes a field common to all shapes. **`reference`** = the point used to start drawing or resizing.

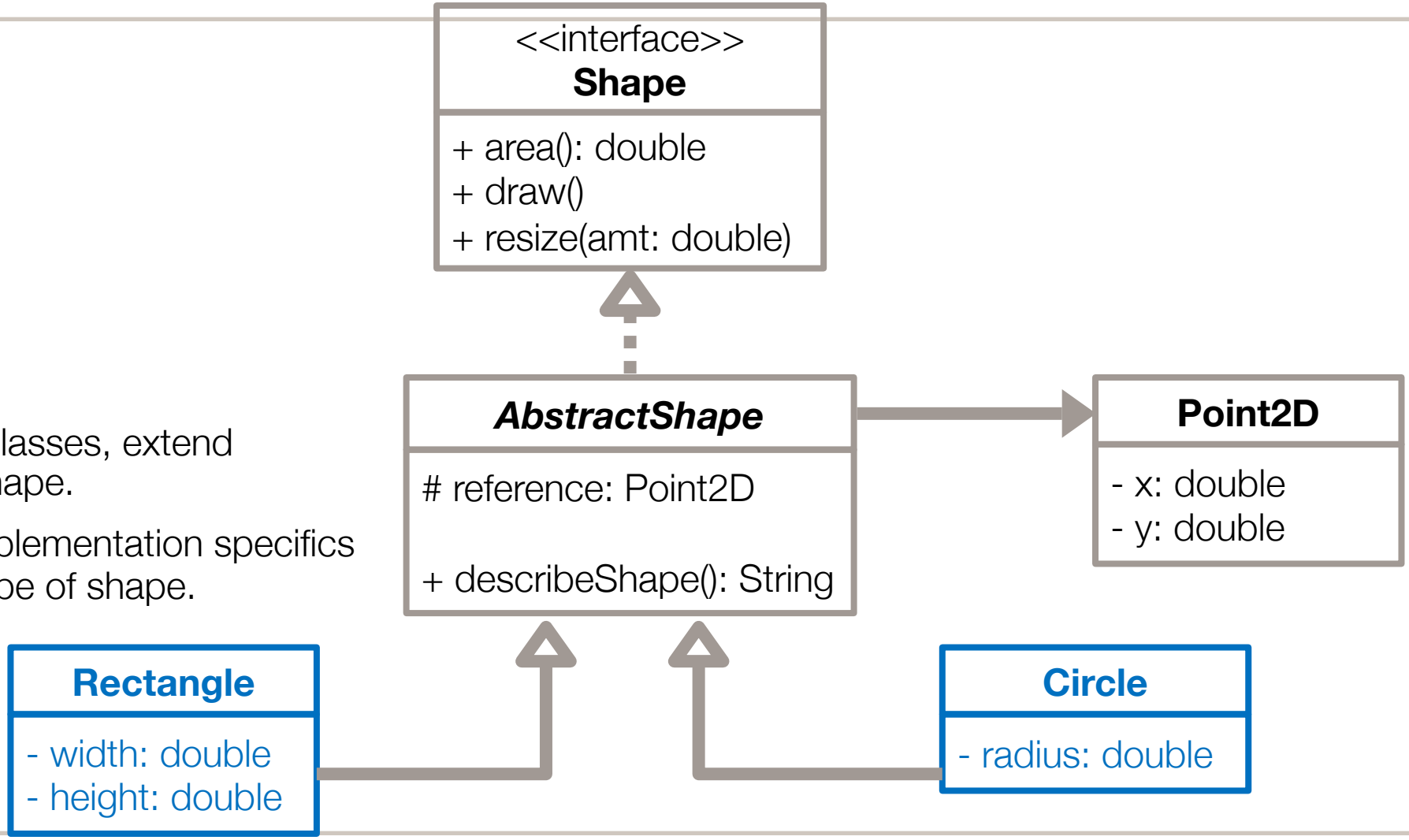


Implements a method that works exactly the same way for all shapes.

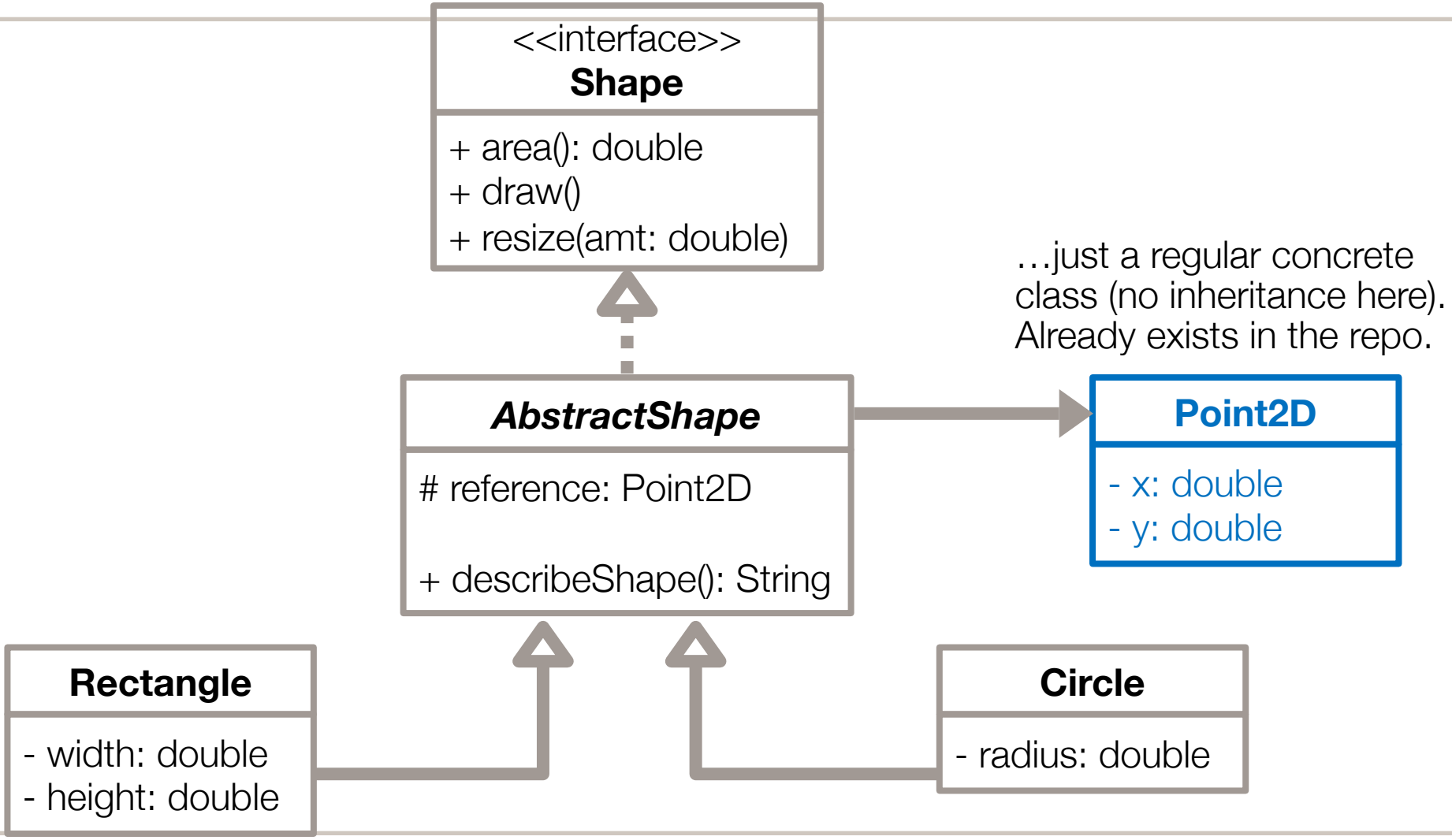
EXAMPLE - MORE ON SHAPES

Concrete classes, extend AbstractShape.

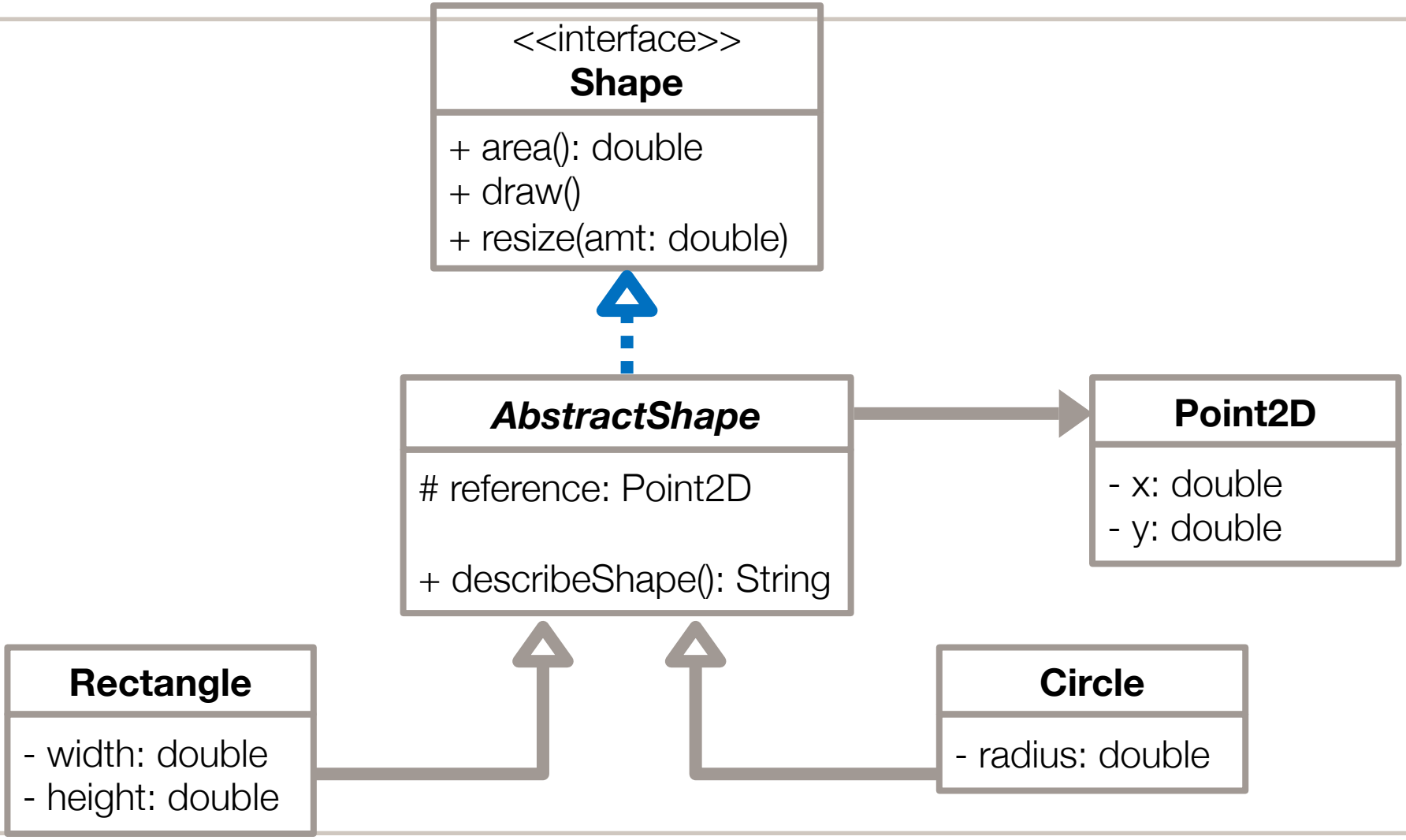
Provide implementation specifics for each type of shape.



EXAMPLE - MORE ON SHAPES

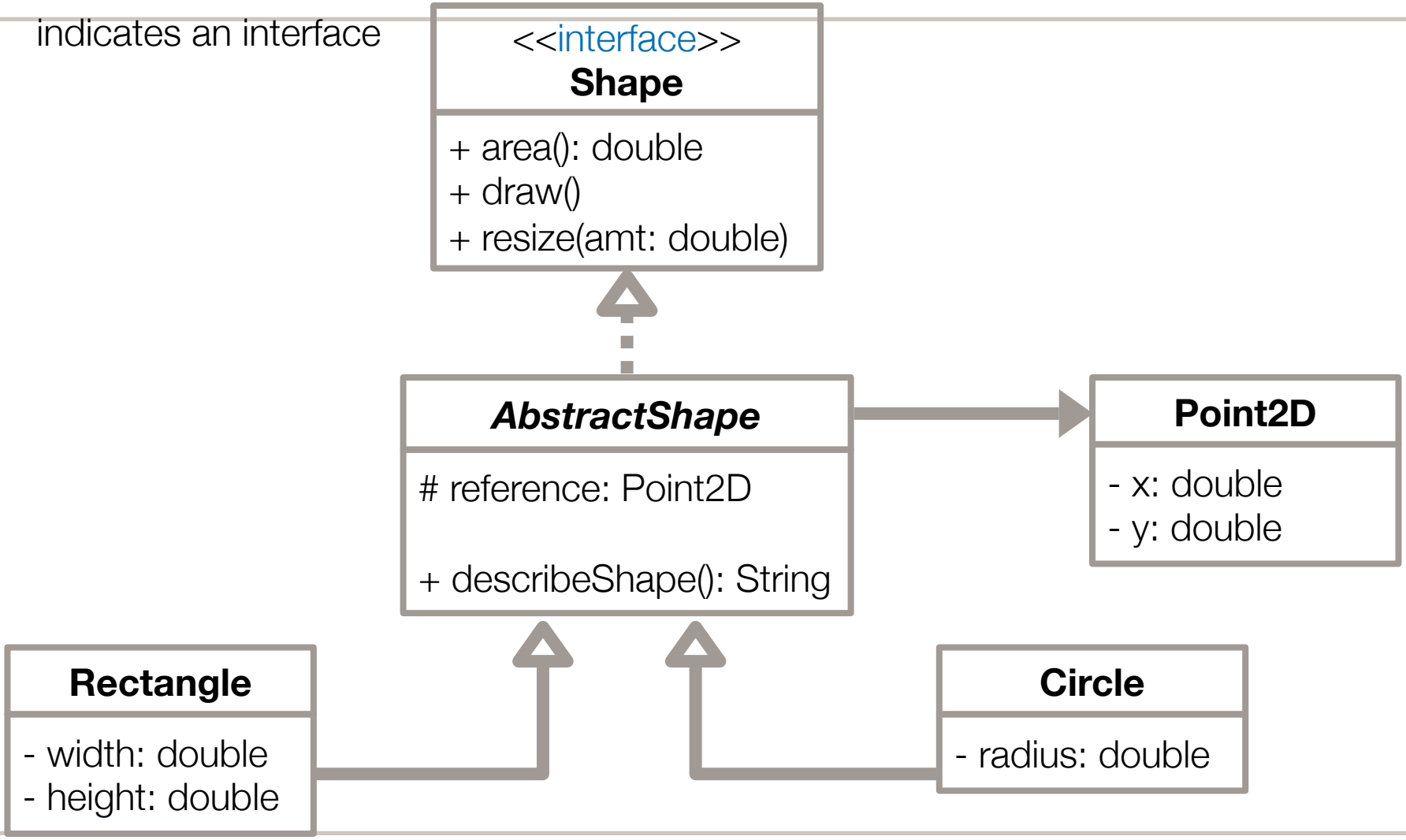


MORE UML CONVENTIONS



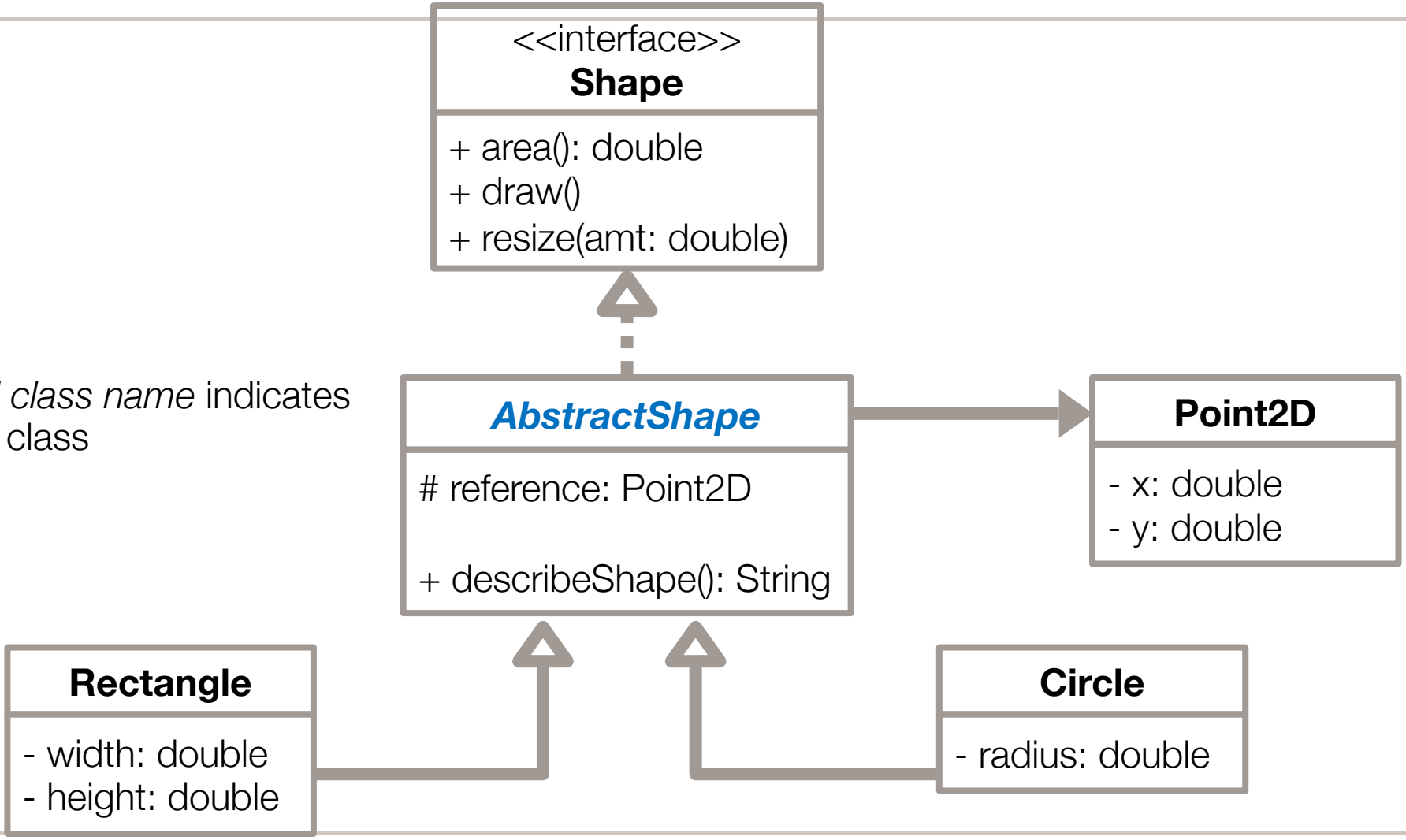
UML CONVENTIONS - INTERFACE

indicates an interface



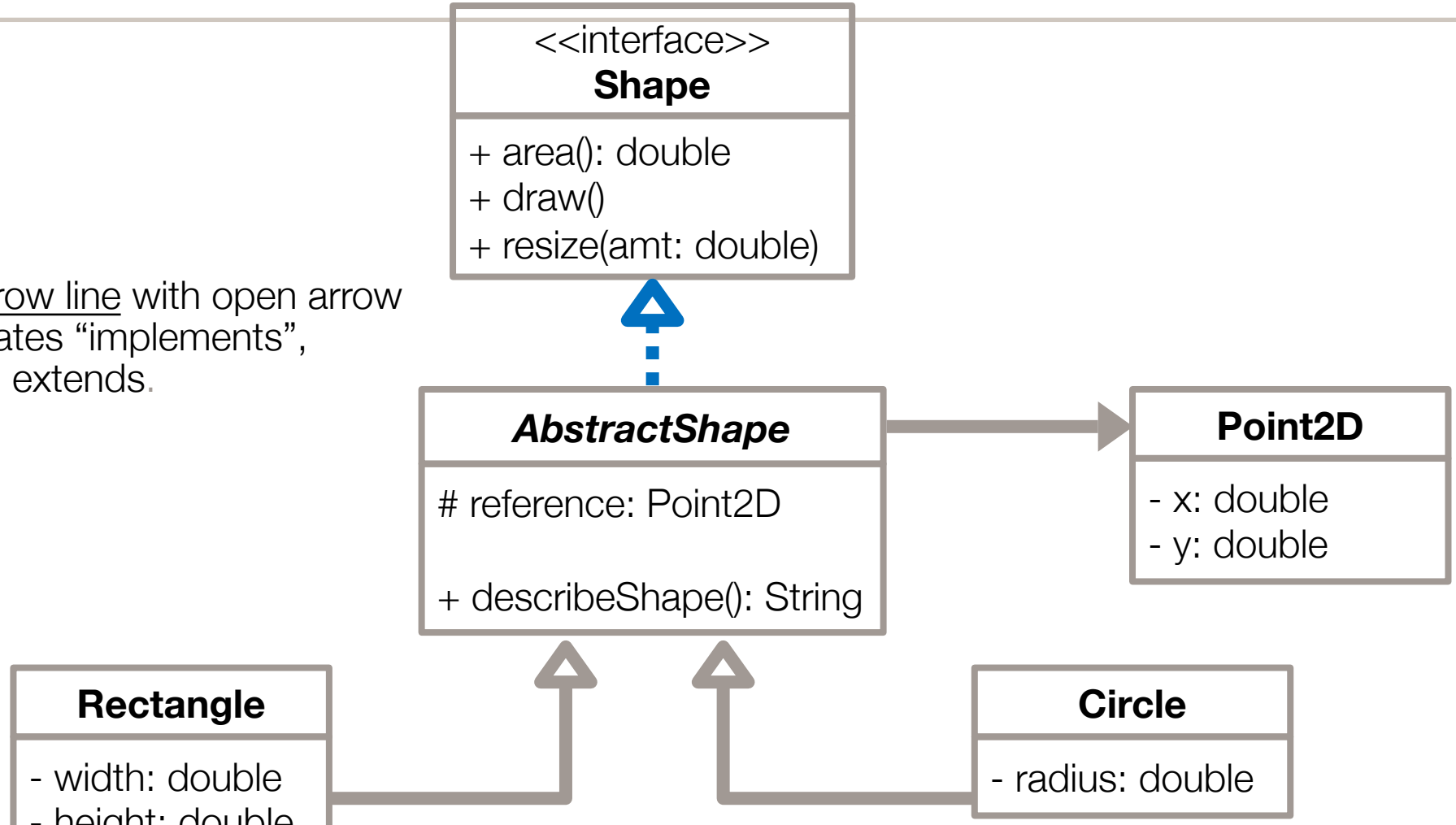
UML CONVENTIONS – ABSTRACT CLASS

Italicized class name indicates abstract class

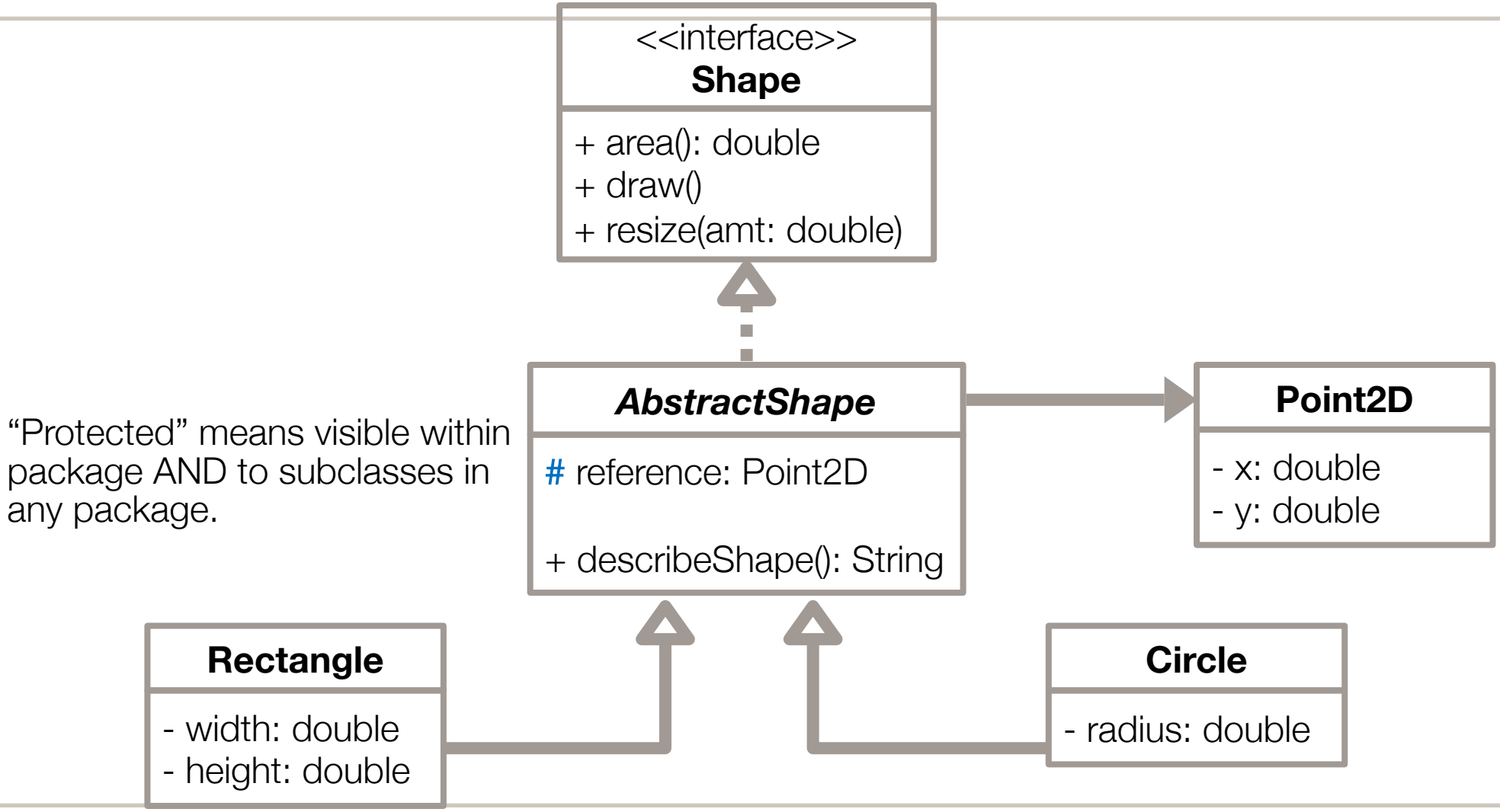


UML CONVENTIONS – IMPLEMENTS

Dashed arrow line with open arrow head indicates “implements”, rather than extends.

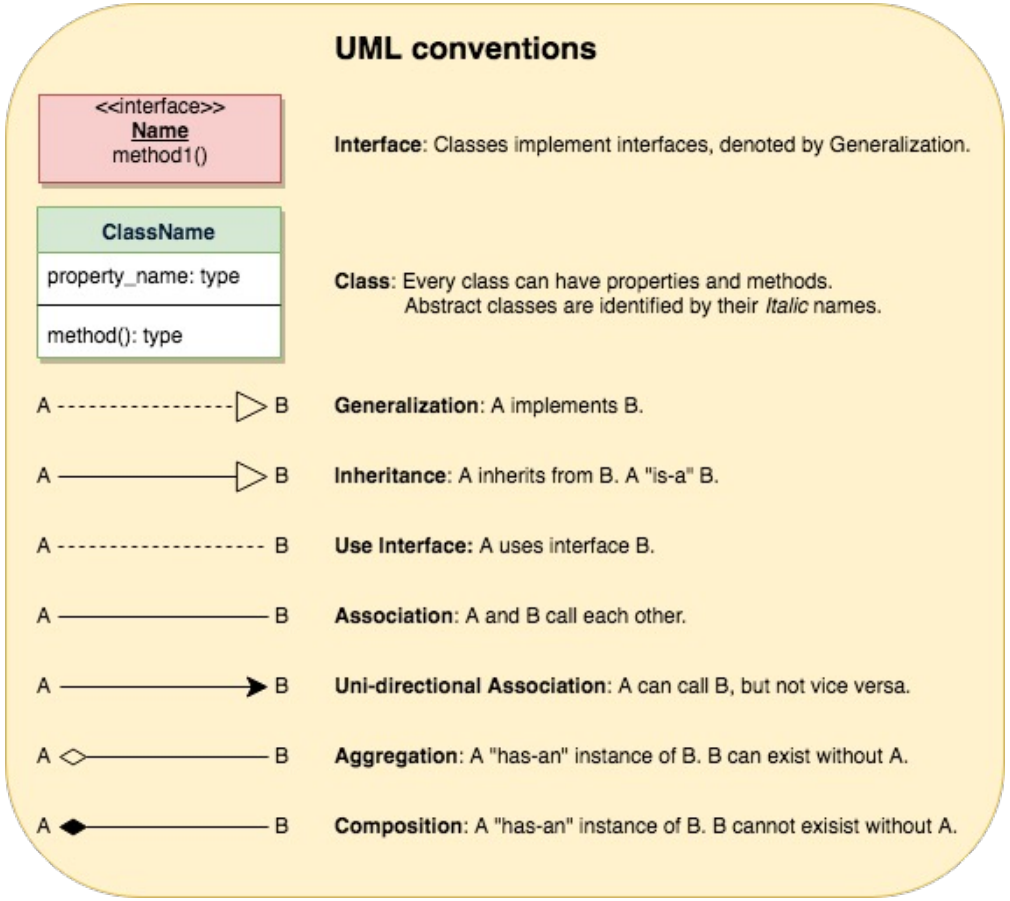


UML CONVENTIONS – PROTECTED ACCESS



"Protected" means visible within package AND to subclasses in any package.

CLASS UML DIAGRAM - SUMMARY



[Figure credit: www.education.io]

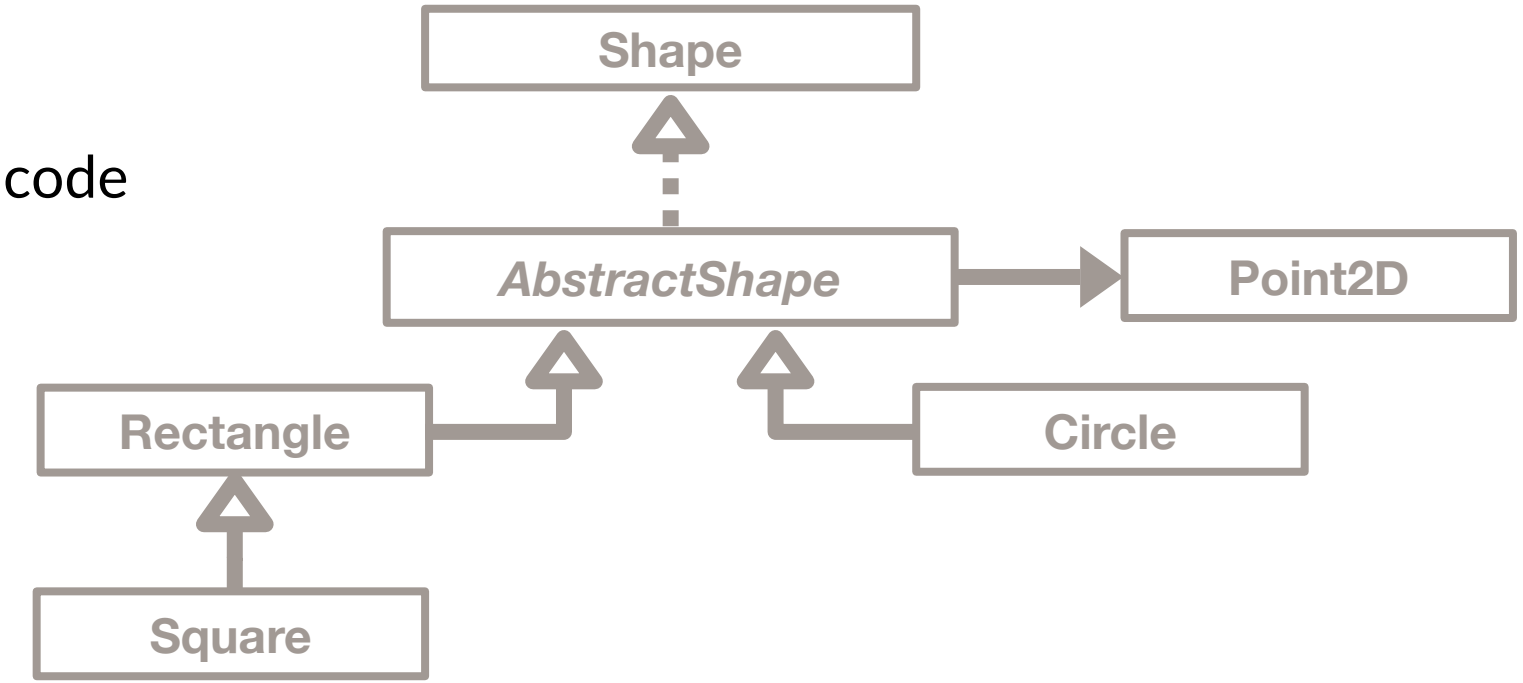
A FEW NOTES

The previous design shows an abstract class implementing an interface and concrete classes extending an abstract class

- Concrete classes can implement interfaces directly
  - You don't need an abstract class in-between
- Abstract classes don't have to implement an interface

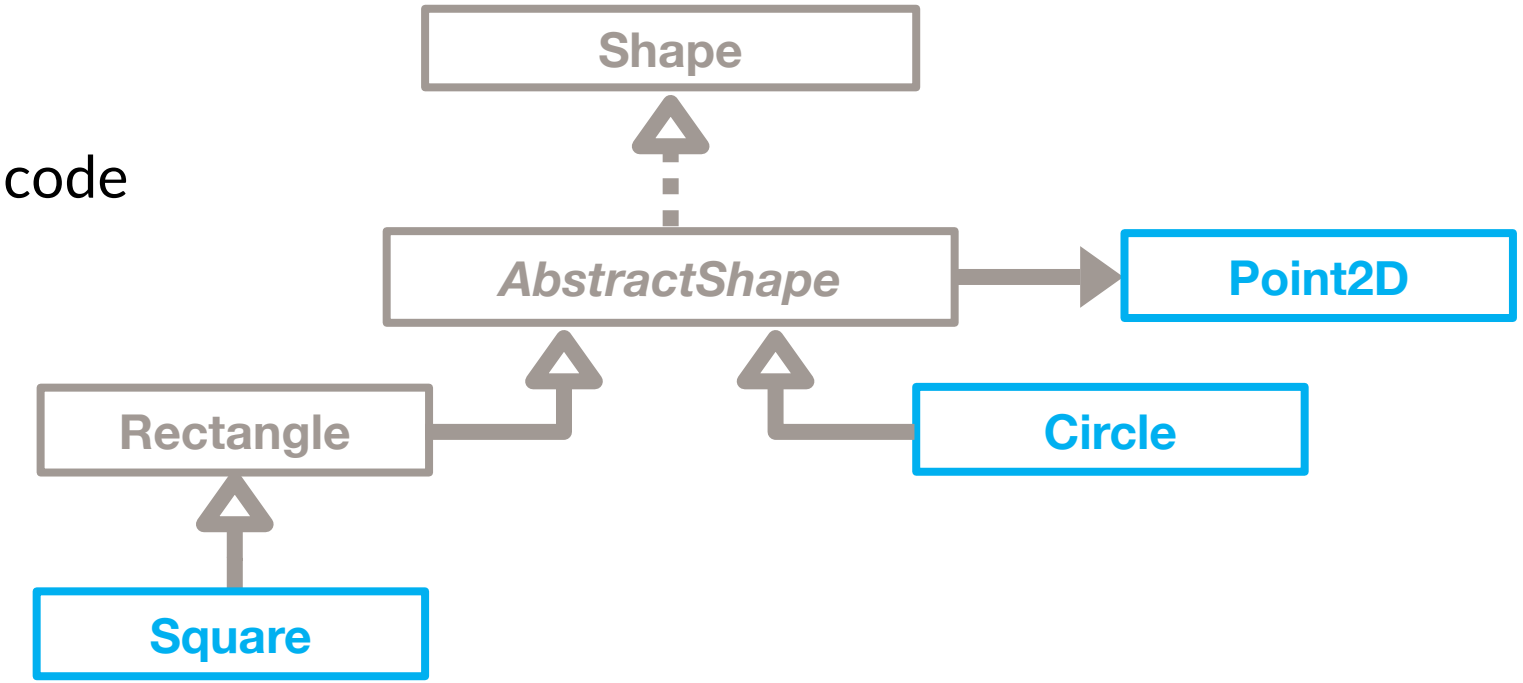
TESTING WITH INHERITANCE

- No need to repeat tests across multiple classes
- Steps:
  - Write tests for concrete classes that don't have subclasses (including inherited methods)
  - Check Jacoco coverage
  - Add tests for uncovered code



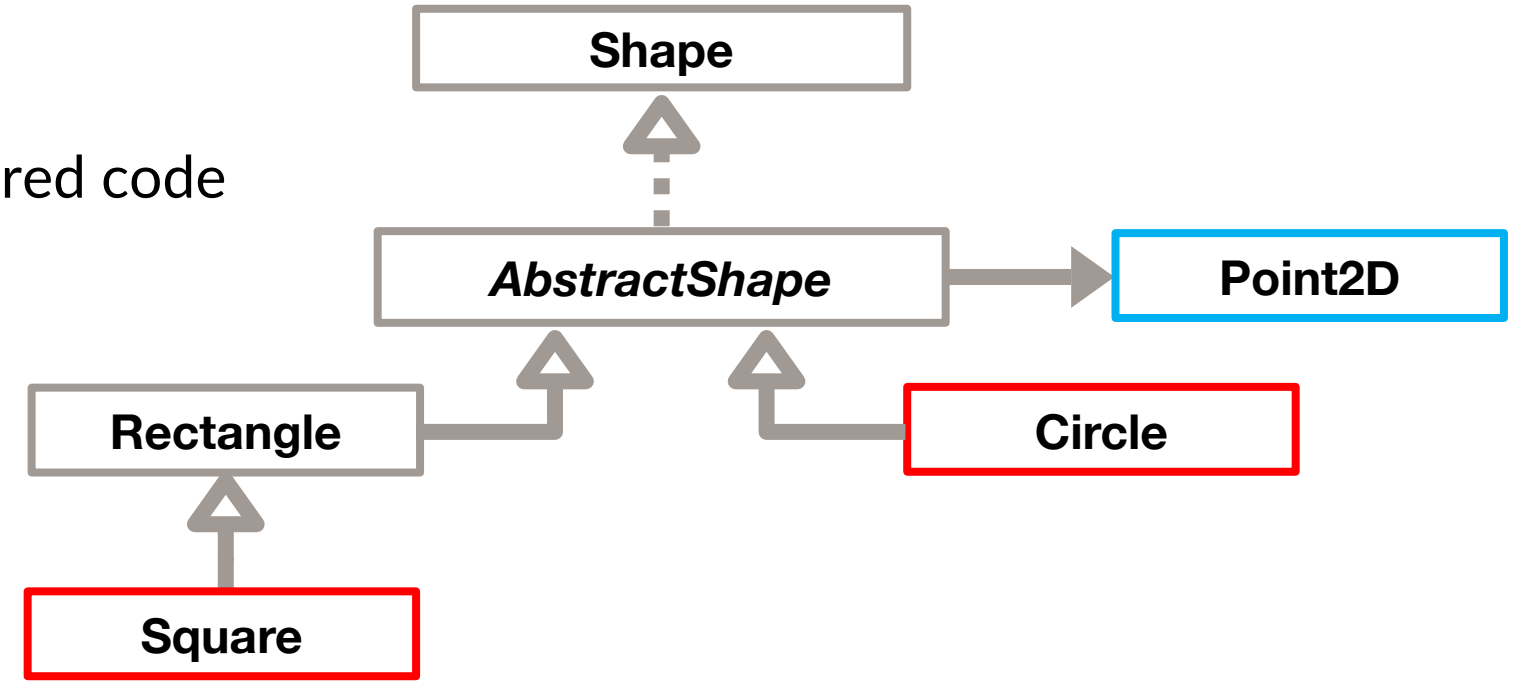
TESTING WITH INHERITANCE

- No need to repeat tests across multiple classes
- Steps:
  - **Write tests for concrete classes that don't have subclasses (including inherited methods)**
  - Check Jacoco coverage
  - Add tests for uncovered code



TESTING WITH INHERITANCE

- No need to repeat tests across multiple classes
- Steps:
  - **Write tests for concrete classes that don't have subclasses (including inherited methods)**
  - Check Jacoco coverage
  - Add tests for any uncovered code



If `SquareTest` covers `Shape` methods `draw()`, `area()`, `resize()`, these methods will not have to be tested for parent classes

INTERFACES AND ABSTRACT CLASSES

- Abstract classes and interfaces cannot be instantiated
  - Intended to be extended by another, concrete class
  - Typically, missing implementations of one or more declared methods defined in the interface

So, which to use – abstract class or an interface?

- Typically, both!

INTERFACES AND ABSTRACT CLASSES

Abstract class	Interface
A class can extend **at most one** superclass (abstract or concrete)	A class can implement **any number** of interfaces
Includes instance variables	No instance variables (in Java 8)
Wider range of modifiers (`private`, `public`, `protected`)	All methods has `public` access modifier by default
Can specify constructors, which subclasses can invoke with keyword `super` (abstract classes still cannot be instantiated)	No constructors (interfaces cannot be instantiate)
Use: to abstract out common states and behavior of children classes	**Use:** a contract, specifying public behavior

[Table credit: Dr. Maria Zontak]

ENUMERATIONS

CS 5004, SPRING 2024 – LECTURE 3

REPRESENTING DATA THAT HAVE FINITE, SPECIFIC VALUES

- Example: we designed a class `Book`, and now we want to add information about the format in which we can buy it (hardcover, paperback, kindle)

- Question: how to represent this information in the Book class?

- Answer: let's use enumerated types

```
public enum TypeOfBook{HARDCOVER, PAPERBACK, KINDLE}
```

WHAT IS AN ENUMERATION?

- Enumeration - a way to represent a set of finite constants
- Represented as an `enum` data type
- What should be an **`enum`**?
  - Days of the week
  - Directions (N, S, E, W)

- **What shouldn't be an `enum`:**
  - Anything that is not finite
  - Anything that could be described as a "type of" something else
  - Anything that has properties/behaviors associated with it

BASIC ENUM STRUCTURE

- `Enum` data types are created in their own files (like a class), with keyword `enum`
- Each field is named in ALL CAPS (because they're always constant)
- Fields are separated by commas, and they don't have data types
- Fields are also not set to equal anything

```
public  enum DayOfWeek
{
MONDAY, TUESDAY,
WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY,
SUNDAY
}
```

USING AN ENUM

- Variables can have an enum data type
- We set the value of an enum variable using:

```
<EnumType>  varName = <EnumType>.<Field>
```

```
DayOfWeek mon = DayOfWeek.MONDAY;
```

THE SWITCH STATEMENT

- An alternative to if-else if-else
- Neater (less typing)
- Only works with enums and a handful of other data types (incl. String)

```
switch(id) {
  case value-one: //is id==value-one?
     [do something 1]
     break;
  case value-two: //is id==value-two?
     [do something 2]
     break;
  ...
  default: //none of the above
     [do something-none-of-the-above]
 }
```

GOOD OOD PRACTICES

CS 5004, SPRING 2024 – LECTURE 3

HOW DO WE REPRESENT ..X..?

- When X is something descriptive, e.g., color, animal species, day of week?

- Do I make X:
  - a `String` field in a class?
  - an `enum` field in a class?
  - a class with its own properties and methods?

HOW DO WE REPRESENT ..X..?

- When X is something descriptive, e.g., color, animal species, day of week?
- Do I make X:
  - a `String` field in a class?
  - an `enum` field in a class?
  - a class with its own properties and methods?

- Factors to consider:
  - Is there a finite and fairly small set of possible values?
  - Is X for information only?
  - ...or are their additional properties/behaviors dependent on the value of X?

IS THERE A FINITE SMALL SET OF POSSIBLE VALUES?

NO - e.g., a person's name, a book title → Use a String field in another class

```
public class  Name {
   private String firstName;
   private String lastName ;

   public Name (String firstName,  String lastName) { ...}
}
```

IS THERE A FINITE SMALL SET OF POSSIBLE VALUES?

YES – e.g., vehicle color, pet species, day of week

→ String field is not a great choice (error prone)
→ Maybe an enum field (if set is fairly small)
→ Maybe a class

More information needed!

- Is X for information only?
- ...or are their additional properties/behaviors dependent on the value of X?

ARE PROPERTIES/BEHAVIORS DEPENDENT ON THE VALUE OF X?

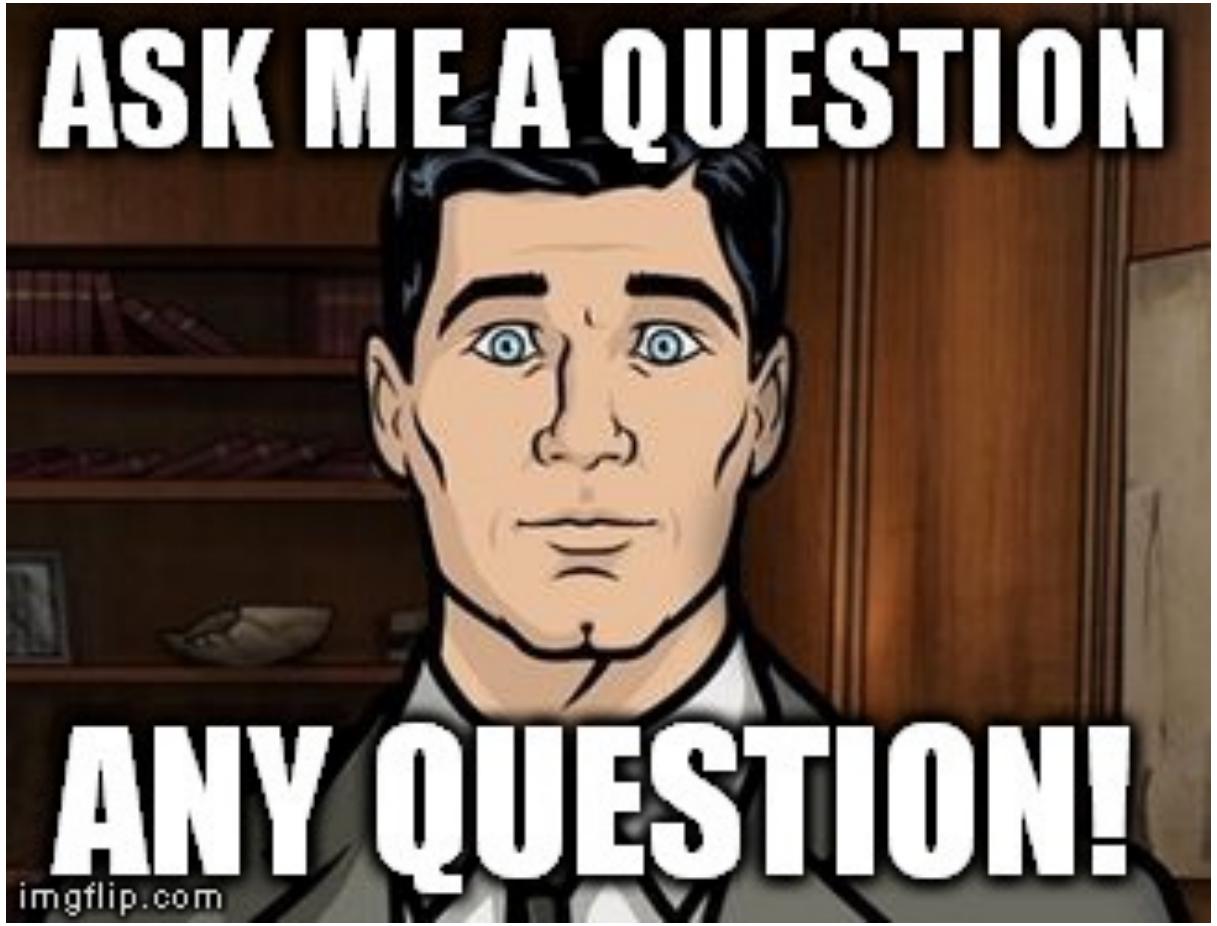
- Might depend on specific situation
  - NO – e.g., vehicle color, day of week (much of the time)
  - An enum field is possibly acceptable

  - YES – e.g., pet species
  - An enum field is NOT the OOD choice
  - A class (or sub class) is usually the most appropriate OOD choice

WOULD YOU DESCRIBE X AS A TYPE OF SOMETHING?

If yes, X should be a class, not a String or an enum!

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)