CS 5004: Lecture 3

Northeastern University, Spring 2021

At the start of every lecture

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

Agenda

- Overriding `Object` methods:
  - `equals, hashCode, toString`
- Inheritance pt 2
- Enums and the `switch` statement
- Good OOD practice:
  - classes vs. enums vs. `String`

Overriding `Object` methods

Review of Lab 2

The built-in Object class

All classes inherit Java's `Object` class:

- built-in classes you use
- custom classes you write
  - Note: you do not need to add `extends Object` to your class definitions!

`Object` methods you should always override* (from now on):

- `boolean equals(Object o)`
- `int hashCode()`
- `String toString()`

*excludes Exception and Test classes

`boolean equals(Object o)`

Indicates whether some object is “equal to” this one. An equivalence relation:

- Reflexive: **`x.equals(x)` → `true`**
- Symmetric: **`x.equals(y)`** iff **`y.equals(x)`**
- Transitive: if **`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`**

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)

Implementing `equals()`

- Use **`@Override`** notation
- Must have the signature: **`public boolean equals(Object o)`**
- You choose how to determine equality but there are 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
- Most of the time, use auto-generated equals

Example: bookexample > Author.java

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Author author = (Author) o;
  return this.name.equals(author.getName()) &&
         this.email.equals(author.getEmail()) &&
         this.address.equals(author.getAddress());
}
```

Example: Author.java

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Author author = (Author) o;
  return this.name.equals(author.getName()) &&
         this.email.equals(author.getEmail()) &&
         this.address.equals(author.getAddress());
}
```

Are they they same object, at the same memory address?

Example: Author.java

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Author author = (Author) o;
  return this.name.equals(author.getName()) &&
         this.email.equals(author.getEmail()) &&
         this.address.equals(author.getAddress());
}
```

Do they have the same class?

Example: Author.java

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Author author = (Author) o;
  return this.name.equals(author.getName()) &&
         this.email.equals(author.getEmail()) &&
         this.address.equals(author.getAddress());
}
```

“cast” o to the appropriate type

Example: Author.java

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  Author author = (Author) o;
  return this.name.equals(author.getName()) &&
         this.email.equals(author.getEmail()) &&
         this.address.equals(author.getAddress());
}
```

Check that the values of all the public fields match

- Depends on other classes overriding equals too

Testing `equals()`

Yes, you should test equals!

- For 100% coverage, test every branch / every possible outcome

Testing `equals()`

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null ||
      getClass() != o.getClass())
    return false;
  Name name = (Name) o;
  return firstName.equals(
              name.getFirstName())
         && lastName.equals(
              name.getLastName());
}
```

```
Name a  = new Name("A", "Name");
Name b  = new Name("A", "Name");
Name c  = new Name("Diff", "Name");
String name  = "A Name";
```

Testing `equals()`

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null ||
      getClass() != o.getClass())
    return false;
  Name name = (Name) o;
  return firstName.equals(
              name.getFirstName())
         && lastName.equals(
              name.getLastName());
}
```

```
Name a  = new Name("A", "Name");
Name b  = new Name("A", "Name");
Name c  = new Name("Diff", "Name");
String name  = "A Name";


assertTrue(a.equals(a));
```

Testing `equals()`

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null ||
      getClass() != o.getClass())
    return false;
  Name name = (Name) o;
  return firstName.equals(
              name.getFirstName())
         && lastName.equals(
              name.getLastName());
}
```

```
Name a  = new Name("A", "Name");
Name b  = new Name("A", "Name");
Name c  = new Name("Diff", "Name");
String name  = "A Name";

assertTrue(a.equals(a));
assertFalse(a.equals(null));
assertFalse(a.equals(name));
```

Testing `equals()`

```
@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null ||
      getClass() != o.getClass())
    return false;
  Name name = (Name) o;
  return firstName.equals(
              name.getFirstName())
         && lastName.equals(
              name.getLastName());
}
```

```
Name a  = new Name("A", "Name");
Name b  = new Name("A", "Name");
Name c  = new Name("Diff", "Name");
String name  = "A Name";

assertTrue(a.equals(a));
assertFalse(a.equals(null));
assertFalse(a.equals(name));
assertTrue(a.equals(b));
assertFalse(a.equals(c));
```

`int hashCode()`

- Computes a unique(ish) integer key from an object, for compatibility with hashing data structures.
- If two objects are equal, they *must* have the same hash code
  - Not guaranteed by **`equals()`**
  - Must override **`hashCode()`** if you override **`equals()`**

Testing `hashCode()`

Yes, you should test hashCode()!

- Test that two equal objects have the same hashCode.

```
Name a  = new Name("A", "Name");
Name b  = new Name("A", "Name");
assertTrue(a.hashCode() == b.hashCode());
```

String toString()

Creates and returns a String representation of an Object

- Like **__str__** in Python classes
- Useful for debugging

Name class example:

```
@Override
public String toString() {
   return this.firstName + " " + this.lastName;
}
```

Testing toString()

Call the **`toString()`** method of your object:

```
Name a  = new Name("A", "Name");
assertEquals("A Name", a.toString());
```

IntelliJ `equals()`, `hashCode()` and `toString()` shortcuts

- Place your cursor in the class name and right-click
- Select Generate > equals() and hashCode() (or toString())
  - Select all the fields you want to include in the calculation

Inheritance pt 2

Interfaces and abstract classes

Inheritance - review

Super class - The class that is inherited from.

- Student *inherits from* Person
- Person is the super class (or base class)

Sub class – The class that inherits from the super class.

- Student is the subclass

```
class Student extends Person {

}
```

Review: inheritance vs. composition

Composition:

- “has a” relationship
- Class A contains fields of type B e.g.
  - Lecture 2 > bookexample, Author *has a* Person

Inheritance:

- “is a” relationship
- Class A extends class B (Class A is a type of class B)
  - Lecture 2 > inheritanceexample, Student *is a* Person & Instructor *is a* Person

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.

```
public class MyClass implements MyInterface {

  void requiredMethod1() {
    // Do something
  }

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

What is an interface?

Classes can ***implement*** interfaces.

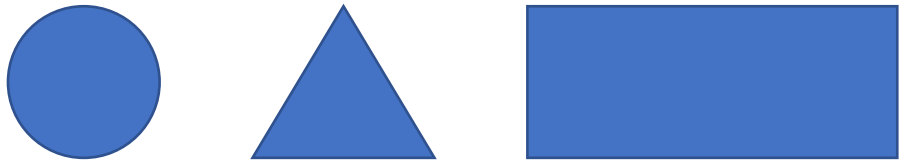
- Classes fill in the implementation details of methods declared in an interface.
- One class can implement multiple interfaces
  - …but *extend* only one super class.

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?

What do the following have in common?



When is an interface useful?

Example: A Shape interface

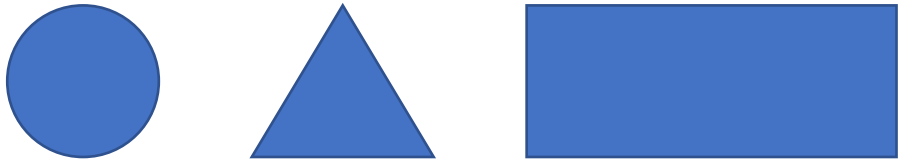
- `area()` – gets the area of a shape.
- `draw()` – draws a shape.
- `resize(double amt)` – resizes a shape by `amt`.

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 {
  // Empty interface called "Shape"
}
```

Basic interface structure

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

Note the keyword, `interface`.

```
public interface Shape {
  // 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

Creating an interface and implementations

Follow along:

- Create the Shape interface.
- Create Rectangle.java and Circle.java, which implement 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:

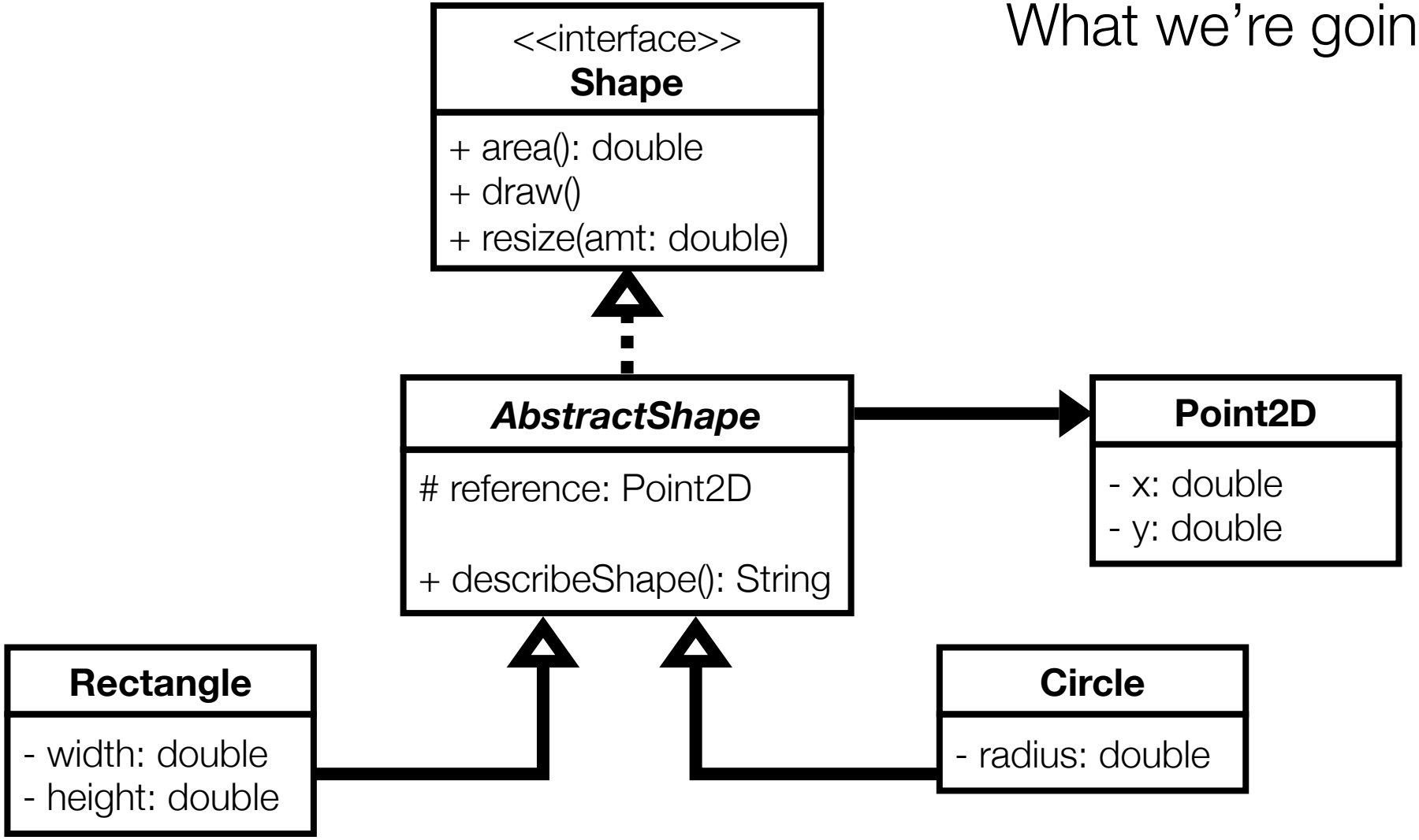
- When you want to provide *some* implementation details common to multiple potential subclasses.

Instead of a concrete class:

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

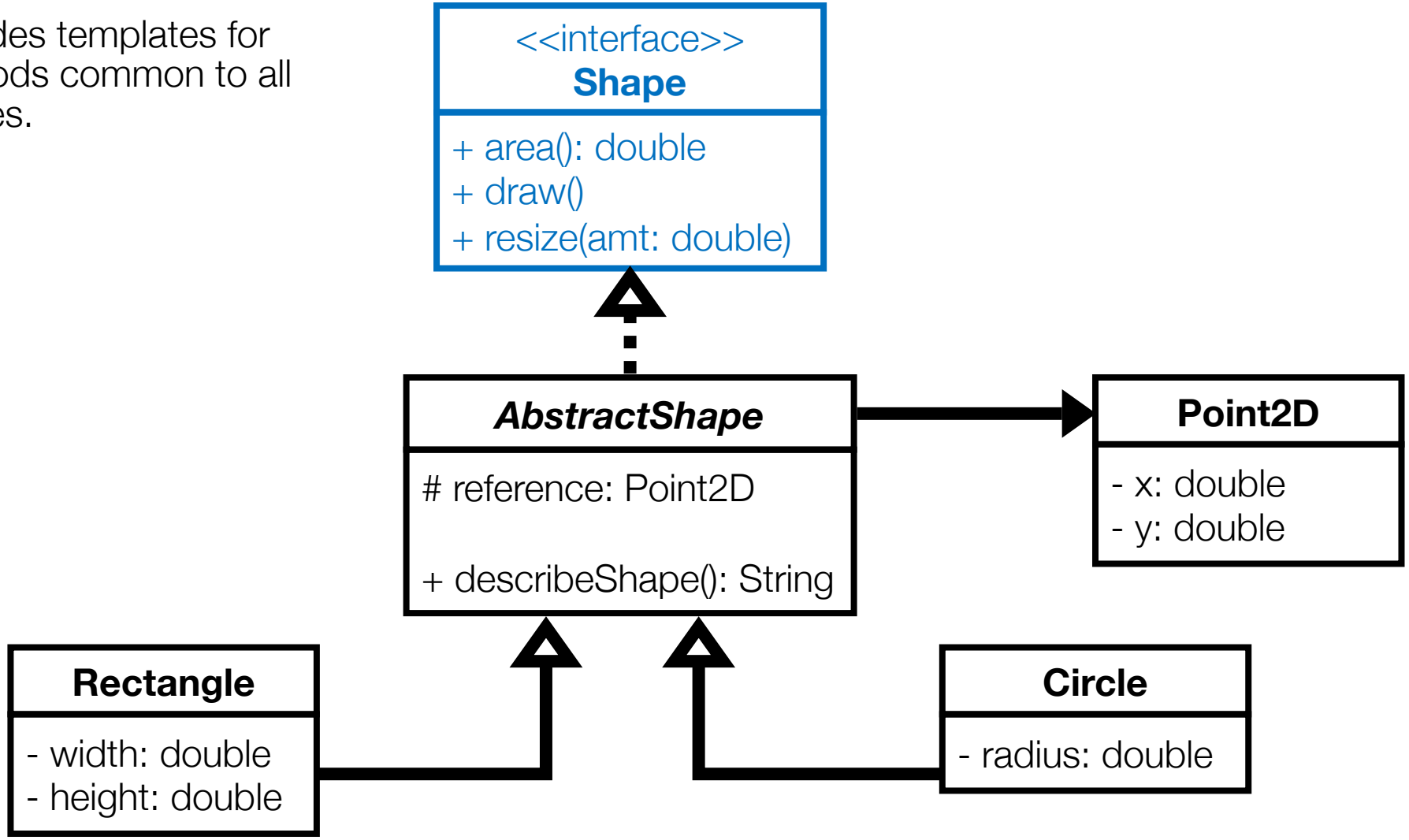
Example - more on shapes

What we're going to build.



Example - more on shapes

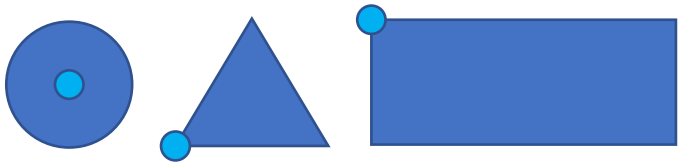
Provides templates for methods common to all shapes.



Example - more on shapes

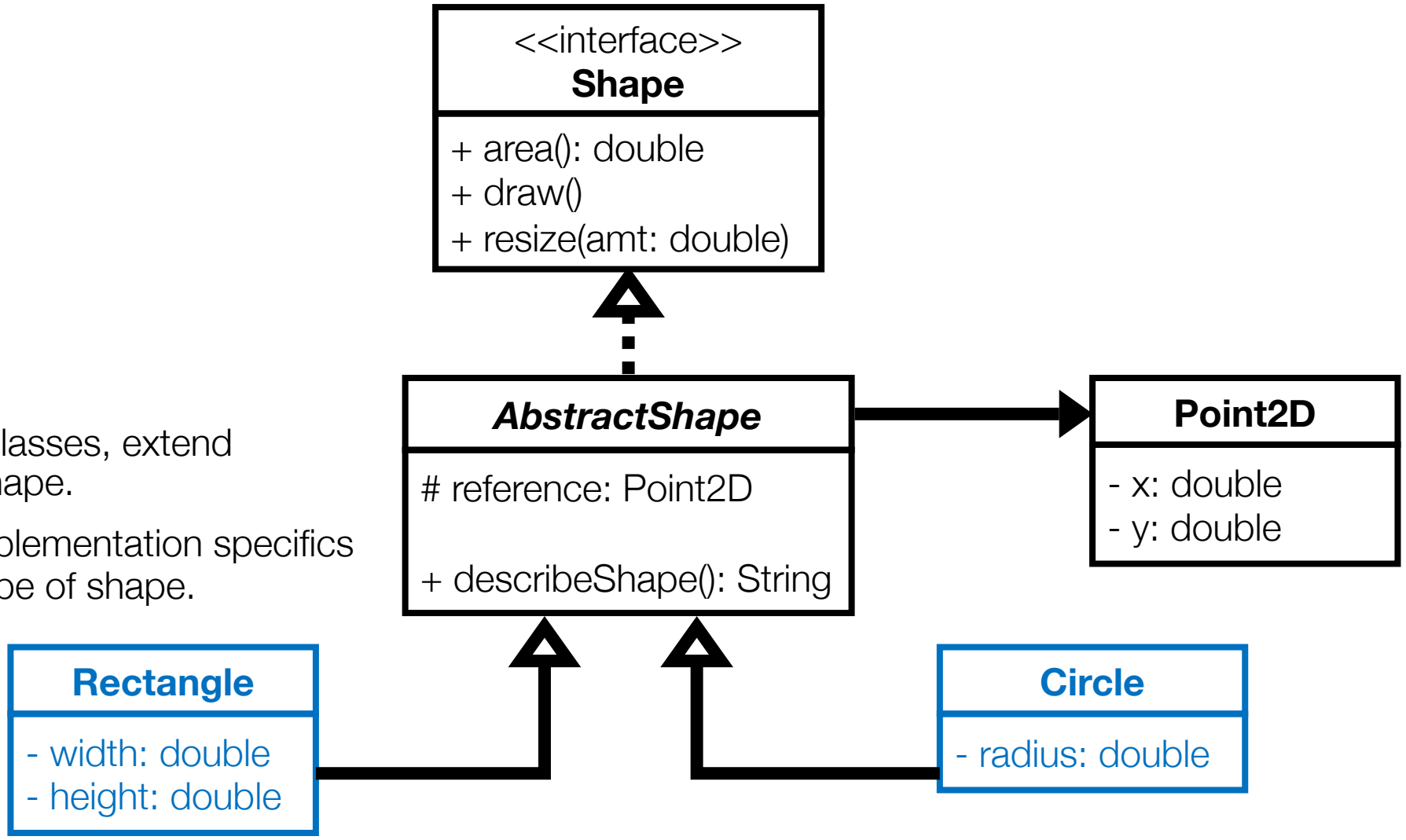
Abstract class, implements Shape.

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.

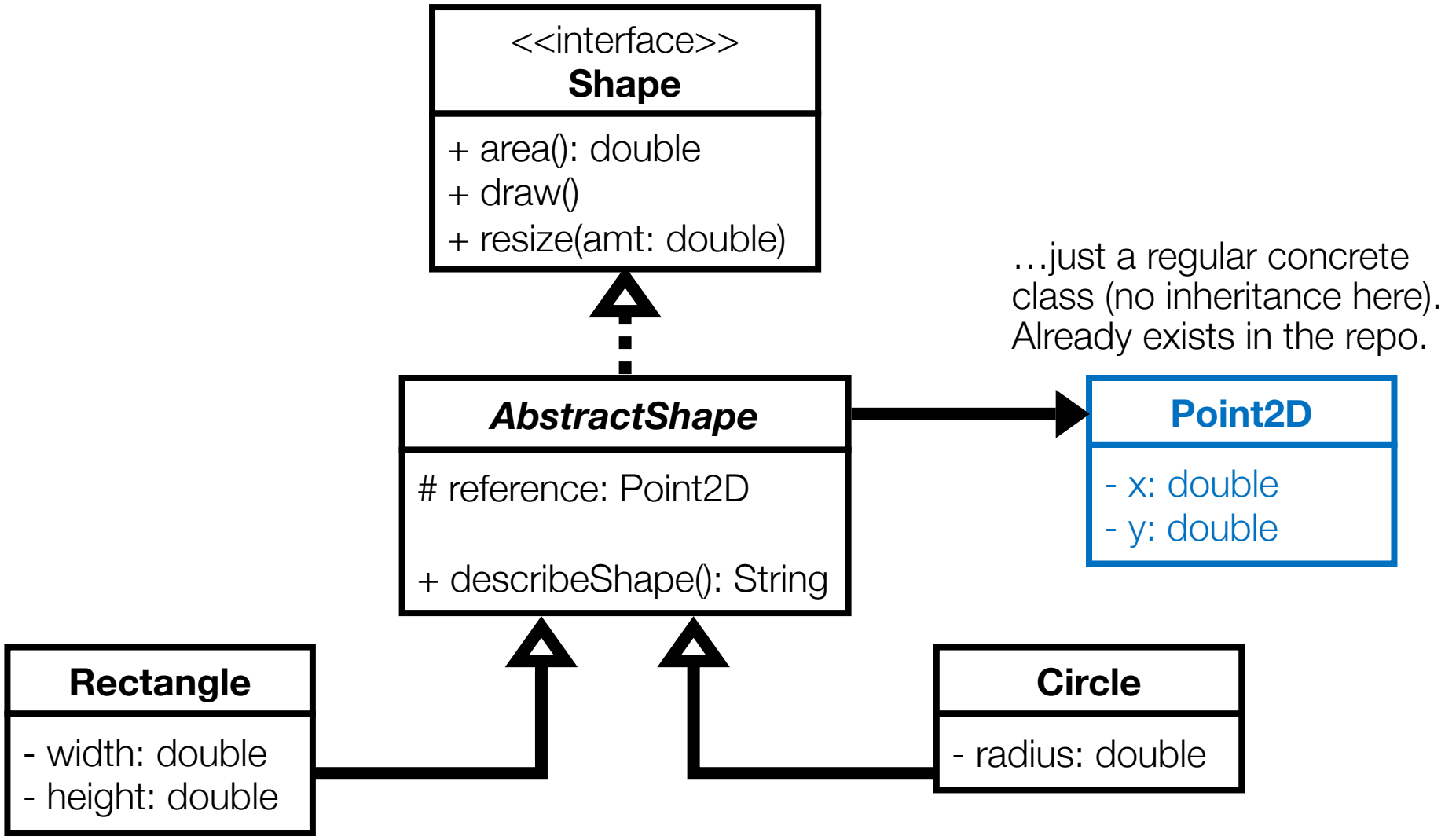
Example - more on shapes



Concrete classes, extend AbstractShape.

Provide implementation specifics for each type of shape.

Example - more on shapes

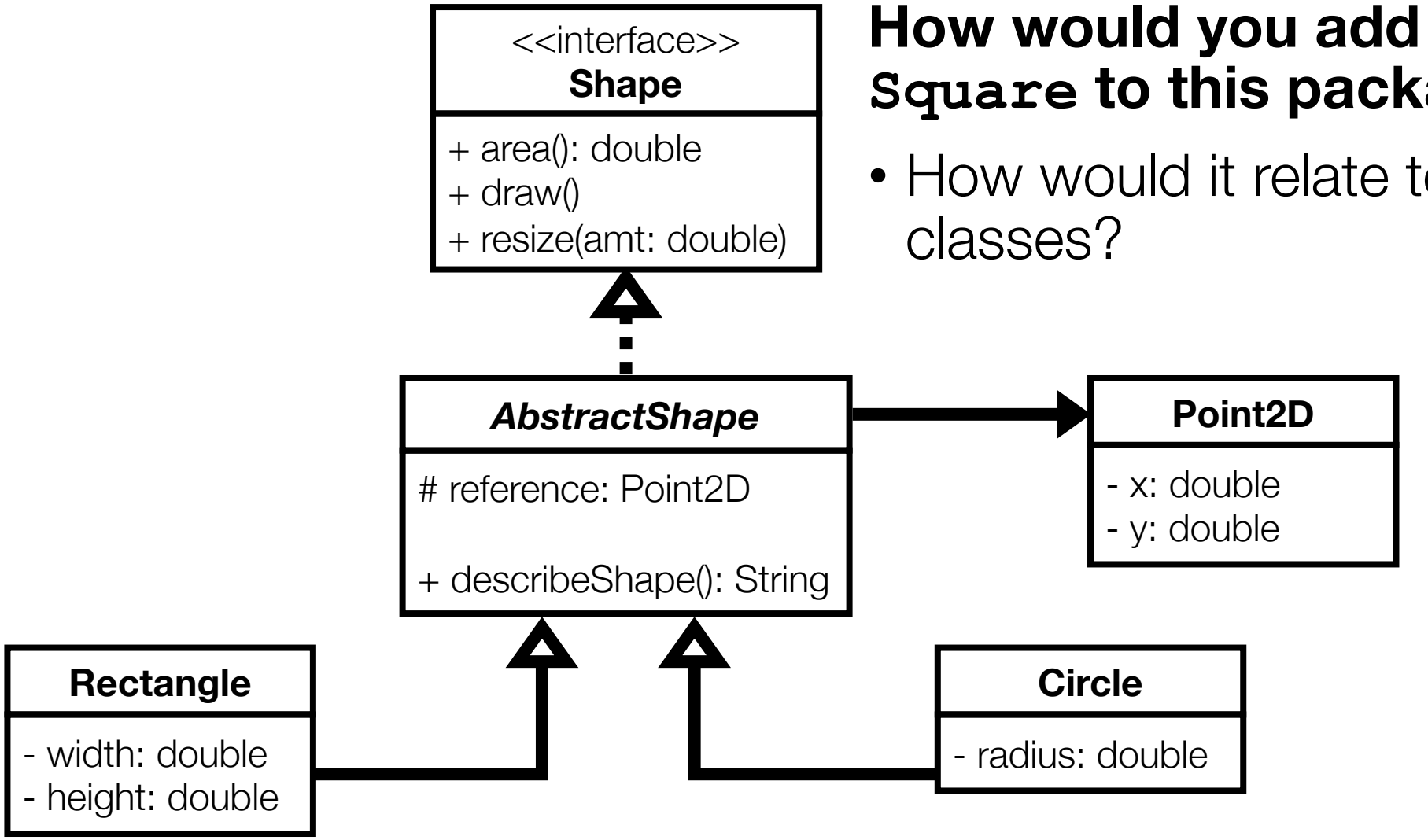


Adding an abstract class

Follow along:

- Create the abstract class, AbstractShape
  - The keyword **`abstract`** indicates that it's an abstract class.
  - AbstractShape will also implement Shape.
- Adjust Rectangle.java and Circle.java to extend AbstractShape.

Exercise 1



How would you add a `Square` to this package?

- How would it relate to other classes?

How would you add a Square to the shapes package?

Answer on the Zoom poll.

A) `Square implements Shape`

B) `Square extends AbstractShape`

C) `Square extends Rectangle`

D) None of the above

Comparison: option 2 vs. option 3

Square extends AbstractShape

```
public Square extends AbstractShape {
  private double width;
  public Square(Point2D ref,
                double width) {
    super(ref);
    this.width = width;
  }
  public double area() {
    return this.width * this.width;
  }
  public void resize(double amt) {
    this.width *= amt;
  }
```

Square extends Rectangle

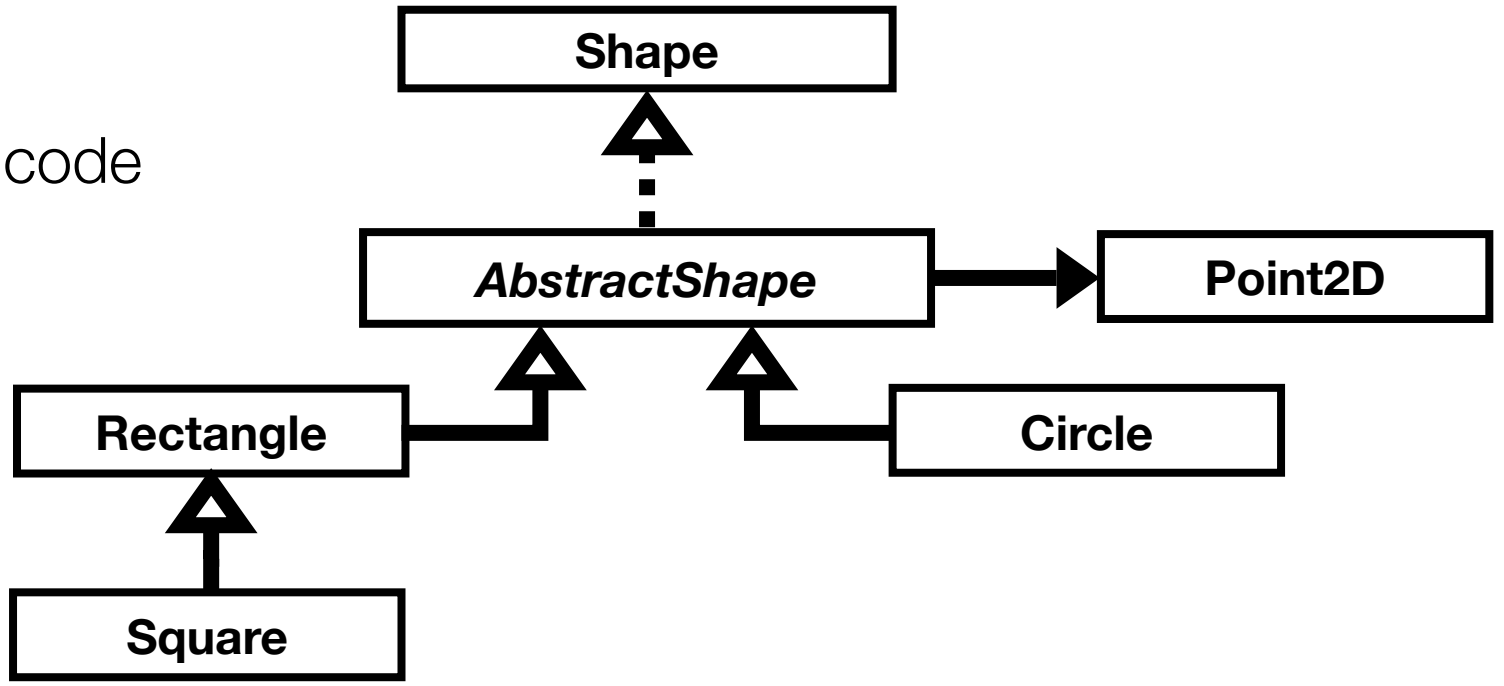
```
public Square extends Rectancle {
  public Square(Point2D ref,
                double width) {
    super(ref, width, width);
  }
}
```

Smart inheritance reduces/ eliminates code duplication

- Best practice = no code duplication
- If you notice code duplication → refactor to remove it

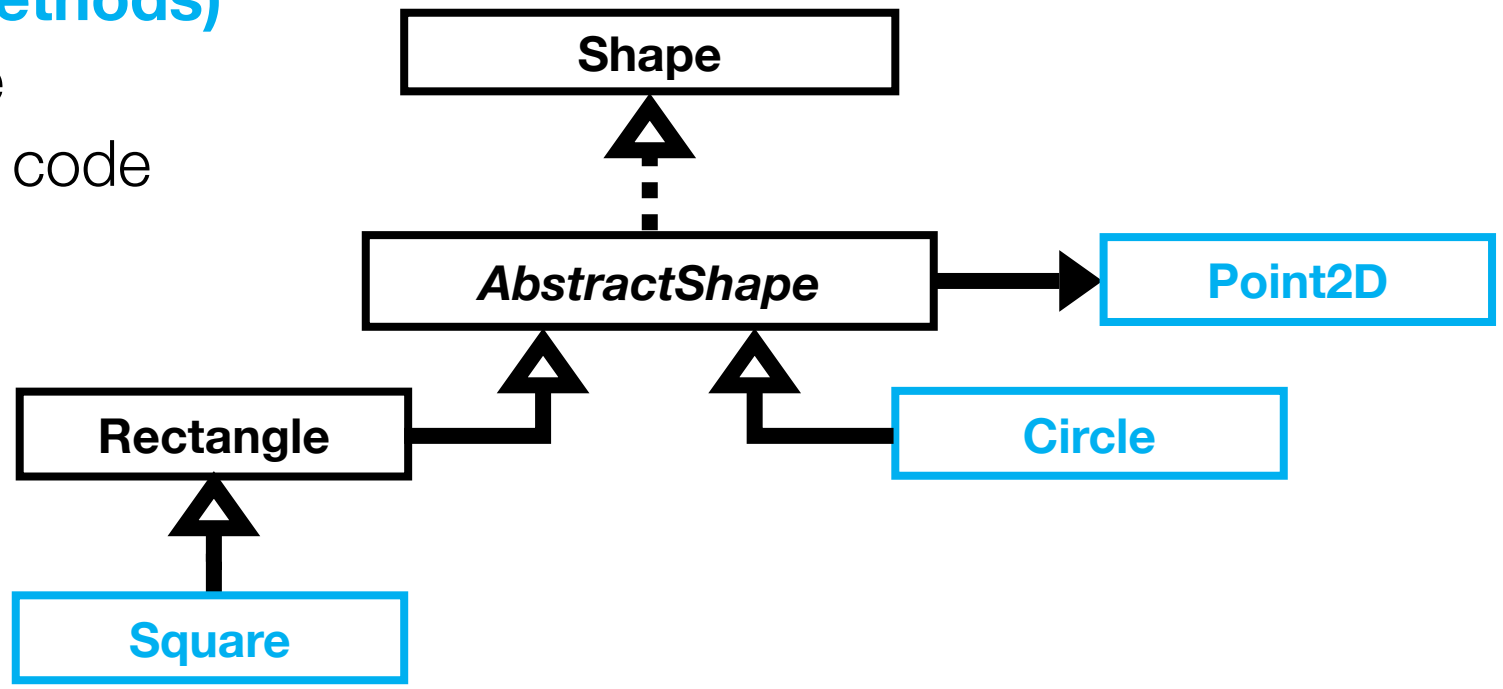
Testing with inheritance

- 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



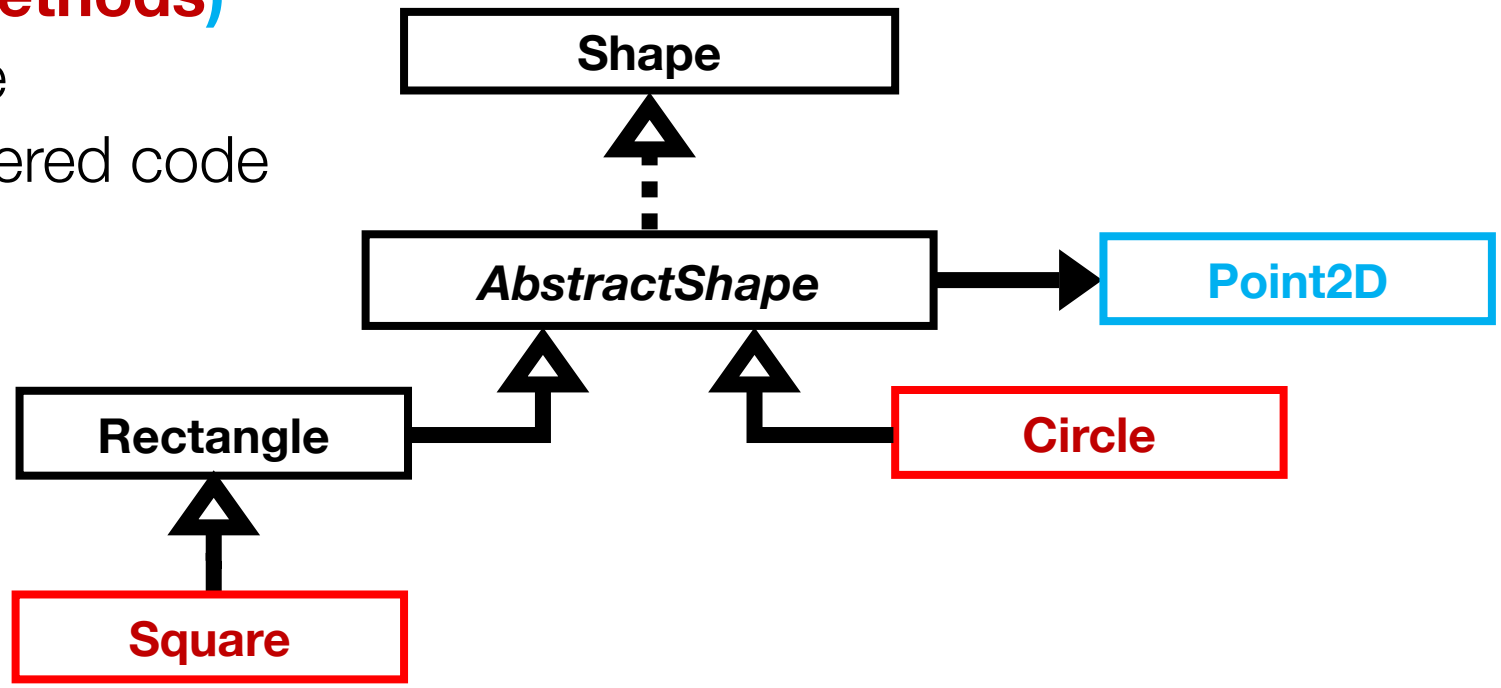
Testing with inheritance

- 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

Break (10 mins)

Enumeration

What is “enumeration”?

A way to represent a set of **finite** constants.

Represented in an **enum** data type.

YES:

- Days of the week.
- Directions (N, S, E, W).

NO:

- Anything that is not finite.
- Anything that could be described as a “type of” something else.
- Anything that has properties/behaviors associated with it.

What is “enumeration”?

A way to represent a set of **finite** constants.

Represented in an **enum** data type.

YES:

- Days of the week
- Directions (N, S, E, W).

NO:

- Height in inches
- Type of vehicle (car, bus, plane)
- Product category
  - Different categories may have different properties/behaviors e.g. labeling requirements, tax rate

Basic enum structure

Enum data types are created in their own files (like a class or interface).

```
public enum MyEnum {
  // An empty enum called "MyEnum"
}
```

Basic enum structure

Enum data types are created in their own files (like a class).

Note the keyword, **`enum`**.

```
public enum MyEnum {
   // An empty enum called "MyEnum"
   // MyEnum is now a data type
}
```

Basic enum structure

Fill in the constants i.e. the specific options/categories for the enum.

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

Basic enum structure

Each field is named in ALL CAPS (because they're always constant)

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

Basic enum structure

Fields are separated by commas.

Note that they don't have data types. Nor are they set to equal anything.

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

Using an enum

Variables can have an enum data type.

```
DayOfWeek mon;
```

Using an enum

Set the value of an enum variable using:

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

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

Using an enum

Set the value of an enum variable using:

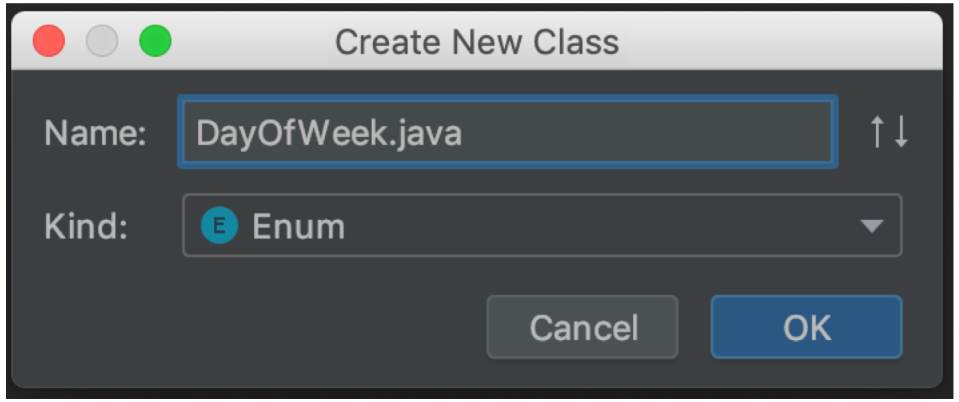
```
<EnumType> varName =
      <EnumType>.<Field>
```

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

The value must be one of the pre-defined constants in your enum definition.

For reference: creating an enum in IntelliJ

- Create a new enum file as you would create a class file:
  - Select the folder where you want to create the file
  - File > New > Class
- In the dialog, change the dropdown from “class” to “enum”:



Enum Javadoc

- Add a Javadoc comment to the enum definition in the same style as a class definition.
- Add Javadoc comments for each value (e.g. MONDAY) in the same style as an instance variable.

Using an enum, “DayOfWeek”

Follow along (lecture 3 > bookstoreexample):

- Implement DayOfWeek enum
- Update Stock.java to use the new enum instead of a String to calculate price after daily discount.

The `switch` statement

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)

The `switch` statement

```
switch (id) {
 case value1: // Is id == value1?
    [do something…];
    break;
 case value2: // Is id == value2?
    [do something 2…];
    break;
 default: // If none of the above…
    [do something else…];
    break;
}
```

- Only checks equality (not <, >, && etc)

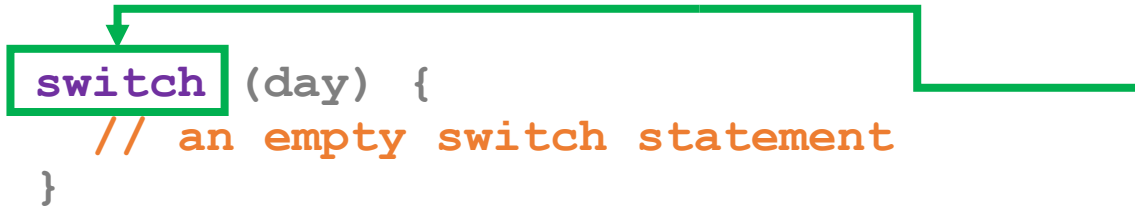
The `switch` statement

```
switch (day) {
 case MONDAY:
   return this.retailPrice * TEN_PERCENT_OFF;

 case TUESDAY:
 case THURSDAY:
   return this.retailPrice * FIFTY_PERCENT_OFF;

 default:
   return this.retailPrice;
}
```

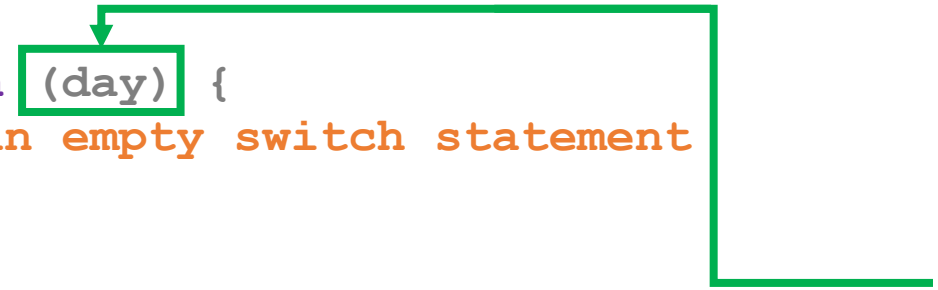
The `switch` statement structure



- Starts with the keyword, **`switch`**

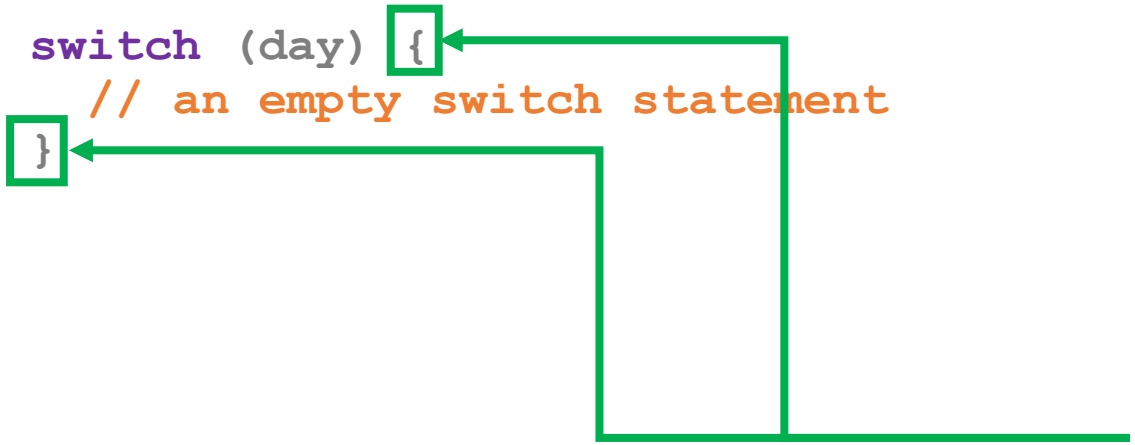
The `switch` statement structure

```
switch (day) {
  // an empty switch statement
}
```



- Starts with the keyword, **`switch`**
- The variable to check in parentheses

The `switch` statement structure



- Starts with the keyword, **`switch`**
- The variable to check in parentheses
- Curly braces to indicate start and end of the statement

The `switch` statement structure

```
switch (day) {
  case MONDAY:
    return ...

  case TUESDAY:
  case THURSDAY:
    return ...

  default:
    return ...
}
```

The switch “block” contains multiple conditional branches.

- Branches start with either **`case`** or **`default`**
- Only one will execute.

The `switch` statement structure - `case`

```
switch (day) {
  case MONDAY:
    return ...

  case TUESDAY:
  case THURSDAY:
    return ...

  default:
    return ...
}
```

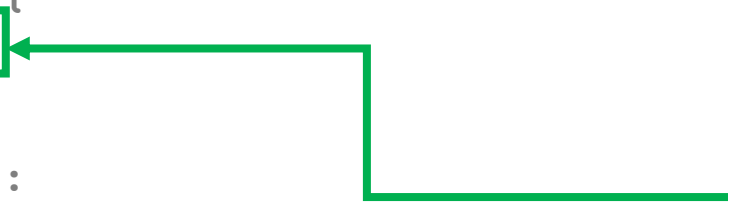
Like an **`if`** or **`else if`** (but not **`else`**)

The `switch` statement structure - `case`

```
switch (day) {
  case MONDAY:
    return ...

  case TUESDAY:
  case THURSDAY:
    return ...

  default:
    return ...
}
```



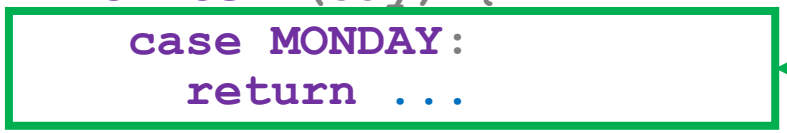
- Like an **`if`** or **`else if`** (but not **`else`**)
- Checks if the value after **`case`** matches the value in parentheses at the start of the **`switch`** statement.
- Note the colon, :

The `switch` statement structure - `case`

```
switch (day) {
  case MONDAY:
    return ...

  case TUESDAY:
  case THURSDAY:
    return ...

  default:
    return ...
}
```



If there's a match, the code in the branch will execute.

- All code after the case/default line and before the next case/default is part of the branch.

The `switch` statement structure - `case`

```
switch (day) {
 case MONDAY:
   return ..
 case TUESDAY:
 case THURSDAY:
   return ...
 default:
   return ...
}
```

If there's a match, the code in the branch will execute.

- Indent for readability.
- If nothing is returned add the following on its own line: **`break;`** – indicates the end of a case, exits the statement.

The `switch` statement structure - `default`

```
switch (day) {
 case MONDAY:
   return ...
 case TUESDAY:
 case THURSDAY:
   return ...
 default:
   return ...
}
```

Like an **`else:`**

- Don't provide a value to match.
- There can only be one default branch.
- Will execute only if none of the cases match.

Good OOD practice

Classes vs. enums vs. String categories

How do I 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 it's own properties and methods?

How do I 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 it's 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?

How do I 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 it's 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 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 there a finite 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

Could the value of X change for a single object?

If X has a finite set of values AND...

It's value will not change, X should probably be a class.

- Definitely a class if other properties are dependent on it!
- Example: pet species – a cat is a *type of* animal, a cat cannot become a dog...

Example: `Car`

A **`Car`** has:

- engine type (gas, hybrid, electric etc)
- a make and model
- a color
- additional features/behavior dependent on engine type e.g.
  - refueling
  - mileage measurement
  - fuel capacity measurement
  - cost to run

Example: `Car`

A **`Car`** has:

- engine type (gas, hybrid, electric etc)
- a make and model
- **a color – String, enum, or class?**
- additional features/behavior dependent on engine type e.g.
  - refueling
  - mileage measurement
  - capacity measurement
  - cost to run

Example: `Car`

A **`Car`** has:

- engine type
- make & model
- **a color – String, enum, or class?** ← Probably not
- additional features/behavior dependent on engine type e.g.
  - refueling
  - mileage measurement
  - capacity measurement
  - cost to run

If available colors may not be finite

If available colors are finite

Example: `Car`

A **`Car`** has:

- **engine type – String, enum, or class?**
- make & model
- a color
- additional features/behavior dependent on engine type e.g.
  - refueling
  - mileage measurement
  - capacity measurement
  - cost to run

Example: `Car`

A **`Car`** has:

- **engine type – String, enum, or class?**
- make & model
- a color
- additional features/behavior dependent on engine type e.g.
  - refueling
  - mileage measurement
  - capacity measurement
  - cost to run

- Describes a type of car e.g. electric is a type of car
- Other behavior is dependent on it
- Value won't change

A symptom that your design is not OOD

If your design requires more than a couple of branches in an if/switch to determine a method outcome → probably not OOD

E.g. in **`Car`** class, if **`engineType`** were an enum:

```
double costToDrive(int miles) {
  switch (this.engineType) {
    case(Engine.GAS):
      return this.getGasPricePerMile() * miles;
    case(Engine.DIESEL):
      return this.getDieselPricePerMile() * miles;
    case(Engine.HYBRID):
      return this.getHybridPricePerMile() * miles;
    ...and on and on... not OOD!
```

Design exercise: discuss, plan, don't code!

Go to Canvas > Modules > today's date > in-class exercises - design

- We will discuss
- ...and look at *one possible* solution

Assignment 3

Looooooong. Start early!

- Design-focused
- Specifically, object-oriented design
- Object-orient design must be object-oriented
  - This means writing classes
  - And making good use of inheritance