CS 5004: Lecture 5

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

- Polymorphism intro
- Static / compile-time polymorphism
- Dynamic / runtime polymorphism
- Static vs. dynamic binding
- Checking object type and casting
- Good OOD practice

Polymorphism intro

Object-oriented design principles

- Encapsulation
- Abstraction
- Information hiding
- **Polymorphism**
  - There can be multiple versions of methods/constructors
  - Objects have the power to act as other objects
- Inheritance

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

"meow"

"woof"

A Cat is also an Animal

A Dog is also an Animal

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

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

Overloading

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 iff 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;
}
```

One parameter, type Point2D

```
public AbstractShape() {
  this.re
}
```

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);  ──────► calls AbstractShape(Point2D reference)
  this.radius = radius;
}

public Circle(double radius) {
  super();  ──────────────► calls AbstractShape()
  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.rad
}
```

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");
```

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.

Walkthrough

lecture-code > Lecture5 > method overloading

- Review overloaded static methods in Arithmetic
- Calling overloaded static methods in ArithmeticTest

Dynamic / run-time polymorphism

Overriding & subtype polymorphism

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);
```

See the two implementations of IListOfStrings

From last week's lecture code:

```java
public class ListOfStringsTest {
    IListOfStrings empty;
    IListOfStrings aList;

    @Before
    public void setUp() throws Exception {
        empty = ListOfStrings.createEmpty();
        aList = ListOfStrings.createEmpty();
        aList.add("A");
        aList.add("B");
    }
```

```java
public class LinkedListOfStringsTest {
    IListOfStrings empty;
    IListOfStrings aList;

    @Before
    public void setUp() throws Exception {
        empty = LinkedListOfStrings.createEmpty();
        aList = LinkedListOfStrings.createEmpty();
        aList.add("A");
        aList.add("B");
    }
```

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

```
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.**

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

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

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

```java
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;
  }
}
```

```java
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

```java
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;
  }
}
```

```java
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

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();
```

Break (10 mins)

Exercise: static vs. dynamic types

- Create a test class for the **`Animal`** interface
- Create one test object per species (**`Cat`**, **`Dog`** etc)
  - All test objects should have **`Animal`** as the static type
- Verify that **`talk()`** returns the appropriate String for each animal.

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 & casting

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

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) {...}
```

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

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

Polymorphism exercise

- In-class exercises # 3, & #4 if there’s time