PIC 20A
Inheritance, Interface, and Polymorphism

David Hyde
UCLA Mathematics

Last edited: April 20, 2020

Outline

Maps application

Imagine writing a program that finds directions from one point to another, and you want to write a `class` for this.

```
public class Direction {
   ...
   public GeoLocation getStartingPoint() { ... }
   public GeoLocation getDestination() { ... }
   public double getDistance() { ... }
   public double getTime() { ... }
}
```

Maps application

If you think about it, directions are best described as smaller paths put together. For example, the directions from UCLA to UC Berkeley can be broken down into:

1. walk a little (UCLA $\rightarrow$ bus stop)
2. Culver City Bus Line 6 (bus stop $\rightarrow$ LAX Lot C Bus Terminal stop)
3. walk a little (LAX Lot C Bus Terminal stop $\rightarrow$ LAX)
4. fly (LAX $\rightarrow$ SFO),
5. walk a little (SFO $\rightarrow$ SFO BART station),
6. BART Yellow Line (SFO station $\rightarrow$ 19th St. Oakland Station),
7. BART Orange line (19th St. Oak. Sta. $\rightarrow$ Downtown Berkeley Sta.)
8. walk a little (Downtown Berkeley Station $\rightarrow$ UC Berkeley).

Writing Direction

So perhaps `Direction` should have a `Path` array as its field.

```
public class Direction {
  private Path[] subPaths;
  ...
  public double getTime() {
    double totalTime = 0;
    for (Path p : subPaths)
      totalTime += p.getTime();
    return totalTime;
  }
}
```

Writing Path

If a Path is a pedestrian path, we could write

```
public class Path {
  public static final double walking_speed = 3;
  public final double distance;
  private final GeoLocation start, dest;
  ...
  public double getTime() {
    return distance/walking_speed;
  }
  public String toString() {
    return "A pedestrian path from " + start +
           " to " + dest + ".";
  }
}
```

Writing Path is a nightmare

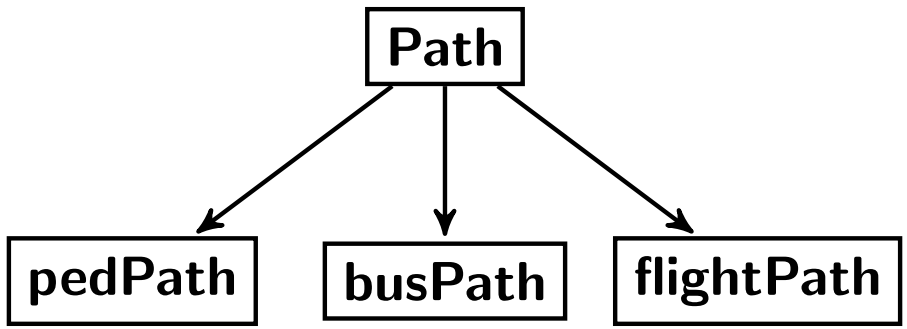
However, Path could be any kind of path, not just a pedestrian path. A bus route is quite different from a pedestrian path. Also, non-pedestrian paths can have toll fees or fairs. Bus or metro paths should contain information on the number of stops...

```
public class Path {
  public static final double walking_speed = 3;
  public final double distance;
  private final GeoLocation start, dest;
  public final boolean isPedestrian, isBus, isPlane,
  public final int fairCost, tollCost;
  public final int numStops;
  ...
  public double getTime() {
    if (isPedestrian) {
      return distance/walking_speed;
    } else if (isPlane) {
      if (isHeadwind)
        return distance/head_wind_airplane_speed;
      else
        return distance/tail_wind_airplane_speed;
```

Inheritance is what you need

Pedestrain paths, bus routes, flights, and other paths should be separate `classes`, but then using them in `class Direction` becomes a pain.

Note the hierarchical structure.



Inheritance is the right tool for this problem.

Outline

Rules of inheritance: basics

Say `class B` *inherits* or *extends* `class A`.

- `A` is called a *superclass*, *base class*, or *parent class*.
- `B` is called a *subclass*, *derived class*, *extended class*, or *child class*.
- `B` inherits `A`'s `public` and `protected` fields and methods (except constructors).
- `B` can have additional fields and methods, in addition to what it inherited from `A`.
- `B` can call `A`'s `public` and `protected` constructors.
- `B` can *override* methods inherited from `A`.
- A reference variable of type `A` can refer to an object of type `A` or `B`.

Example: class Part

Say you want to write a `class Part` that represents mechanical parts.

```
public class Part {
  public final double cost, weight;
  public Part(double cost, double weight) {
    this.cost = cost; this.weight = weight;
  }
  public String toString() {
    return "This is a part with cost " + cost
         + "and weight " + weight + ".";
  }
}
```

Example: class Screw

A screw is a mechanical part.

```
public class Screw extends Part {
  public final double size;
  public Screw(double cost, double weight,
               double size) {
    super(cost, weight);
    this.size = size;
  }
  public String toString() {
    return "A screw of size " + size
         + " and cost " + cost + ".";
  }
}
```

Example: class Screw

Screw inherits Part

```
public class Screw extends Part
```

With super, the constructor of Screw calls the constructor of Part and then initializes its field size.

```
  public Screw(double cost, double weight,
               double size) {
    super(cost, weight);
    this.size = size;
  }
```

Example: class Screw

The `toString` method of `Screw` overrides the `toString` inherited from `Part`.

```
public String toString() {
  return "A screw of size " + size
       + " and cost " + cost + ".";
}
```

`Screw` can access `cost` since its inherited from `Part`.

Example: using Screw

You can use `Screw` like any `class`, and you can access the inherited `public` fields `cost` and `weight`.

```
public class Test {
  public static void main(String[] args) {
    Screw s1 = new Screw(5, 2, 17);
    System.out.println(s1);
    System.out.println(s1.cost);
    System.out.println(s1.weight);
    System.out.println(s1.size);
  }
}
```

Example: Part can refer to many parts

A reference variable of type `Part` can refer to a `Part` or a `Screw`.

```
public class Test {
  public static void main(String[] args) {
    Part[] part_arr = new Part[3];
    part_arr[0] = new Part(1, 10);
    part_arr[1] = new Screw(5, 2, 17);
    part_arr[2] = new Bolt(12, 13, 14);
    for (int i=0; i<3; i++)
      System.out.println(part_arr[i]);
  }
}
```

Each `println` calls the version of `toString` that belongs to each `class`.

java.lang.Object

All objects (except `java.lang.Object`) that don't explicitly inherit another `class` inherits `java.lang.Object`.

This is where `toString` comes from; it is inherited from `Object`.

`equals` is another useful function inherited from `Object`. We will see how to override `equals`.

`https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`

protected vs. private

A `protected` member is accessible by a derived `class`, but not from the outside world. (`class`es within the same `package` can access `protected` members as well.)

A `private` member is only accessible within the `class`. However, a derived `class` can indirectly access a `private` member through an inherited method.

protected vs. private

```
public class A {
  private int field;
  ...
  protected int getField() { return field; }
}
```

```
public class B extends A {
  ...
  public void someMethod() {
    System.out.println(getField()); //okay
    //System.out.println(field); //error!
  }
}
```

Ordering access modifiers

The four access modifiers, in order of least to most restrictive are:
`public` > `protected` > package-private > `private`

Modifier	`class`	`package`	subclass	world
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
package-private	Y	Y	N	N
`private`	Y	N	N	N

Rules of inheritance: fields

- `public` fields are inherited as `public` fields.
- `protected` fields are inherited as `protected` fields.
- `private` and package-private fields are not inherited.
- Inherited fields cannot be overridden. They can only be hidden.

These rules apply to `static` and instance fields.

Rules of inheritance: methods

- A `public` method is inherited as a `public` method.
- a `protected` method is inherited as a `protected` method.
- `private` and package-private methods are not inherited.

These rules apply to `static` and instance methods.

Rules of inheritance: instance methods

Say `class B` inherits `class A`.

- When `B` defines a method with the same signature as an instance method of `A`, the inherited method is overridden.
- When `B` overrides an instance method of `A`, it can widen but not narrow the access modifier.
  - If overridden, a `public` method must be overridden as a `public` method.
  - If overridden, a `protected` method can be overridden as `public` or a `protected` method.
- `B` doesn't inherit and thus cannot override `private` and package-private methods. (If `B` defines a method with the same signature as a `private` or package-private method of `A`, the two methods bare no relationship to each other.)

Rules of inheritance: static methods

Say `class B` inherits `class A`.

- When `B` defines a method with the same signature as an `static` method of `A`, the inherited method is hidden.
- `static` and instance methods are separate and one cannot override or hide another.
  - An instance method of `B` cannot override or hide a `static` method of `A`. This causes a compile time error.
  - An `static` method of `B` cannot override or hide an instance method of `A`. This causes a compile time error.

Rules of inheritance: only members are inherited

In Java, a *member* of a `class` are its methods and fields. `public` and `protected` members are inherited.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

Hiding

For example, consider the base class A

```
public class A {
  public int field = 0;
  public static int field_s= 0;
  public int fn() { return 0; }
  public static int fn_s() { return 0; }
}
```

and derived class B

```
public class B extends A {
  public int field = 1; //hide
  public static int field_s= 1; //hide
  public int fn() { return 1; } //override
  public static int fn_s() { return 1; } //hide
}
```

Hiding

```
public class Test {
  public static void main(String[] args) {
    A a1 = new A();
    System.out.println(a1.field); //output 0
    System.out.println(a1.field_s); //output 0
    System.out.println(a1.fn()); //output 0
    System.out.println(a1.fn_s()); //output 0
  }
}
```

The members of `A` are called. (The fact that `B` inherits `A` is irrelevant here.)

Hiding

```
public class Test {
  public static void main(String[] args) {
    B b1 = new B();
    System.out.println(b1.field); //output 1
    System.out.println(b1.field_s); //output 1
    System.out.println(b1.fn()); //output 1
    System.out.println(b1.fn_s()); //output 1
  }
}
```

The members of `B` (whether they override or hide inherited methods) are called.

Hiding

```
public class Test {
  public static void main(String[] args) {
    A a2 = new B();
    System.out.println(a2.field); //output 0
    System.out.println(a2.field_s); //output 0
    System.out.println(a2.fn()); //output 1
    System.out.println(a2.fn_s()); //output 0
  }
}
```

Because a2 is a reference of type `A`, the hidden members of `A` are called. Regardless of the reference type, the overriding method of `B` is called.

Avoid hiding

Hiding usually causes confusion and brings little benefit. It's generally best to avoid name conflicts.

super

Say `class B` inherits `class A`.
Access overriden or hidden members of `A` within `B` with `super`.

```
public class B extends A {
  public int field = 1; //hide
  public static int field_s= 1; //hide
  public int fn() { return 1; } //override
  public static int fn_s() { return 1; } //hide
  public void accessSuper() {
    System.out.println(super.field); //output 0
    System.out.println(super.field_s); //output 0
    System.out.println(super.fn()); //output 0
    System.out.println(super.fn_s()); //output 0
  }
}
```

Overriding vs. hiding methods

Say `class B` inherits `class A`. You override instance methods while you hide `static` methods.
Within `B` you can access inherited methods of `A` whether they are overridden or hidden or not with `super`. From the outside, you can access hidden methods but not overridden methods.

```
public class Test {
  public static void main(String[] args) {
    B b1 = new B();
    A a1 = b1;
    System.out.println(b1.fn()); //output 1
    System.out.println(a1.fn()); //output 1
    System.out.println(a1.fn_s()); //output 0
    System.out.println(b1.fn_s()); //output 1
    System.out.println(((A) b1).fn_s());
  }                                    //output 0
}
```

Overriding vs. hiding methods

Hidden methods are hidden but accessible.

Overriden methods are inaccessible from the outside. This is the whole point of inheritance!

```
public class Direction {
  private Path[] subPaths;
  ...
  public double getTime() {
    double totalTime = 0;
    for (Path p : subPaths)
      totalTime += p.getTime();
    return totalTime;
  }
}
```

You want `p.getTime()` to use the overriding method to compute the time based on what kind of path `p` refers to.

Hiding, shadowing, obscuring

Shadowing and *obscuring* happens when names conflict and ambiguity arises. They are similar to hiding.

```
public class A {
  private double field;
  public A(double field) {
    this.field = field; //argument shadows field
  }
}
```

(This use of shadowing is okay.)

Hiding, shadowing, obscuring

```
import java.lang.Math;

public class Test {
  public static void main(String[] args) {
    String Math = "Hello";
    //variable name obscures class name
    //System.out.println(Math.PI); //error!
  }
}
```

We won't formally distinguish these 3 types of name conflicts.
Just be mindful that name conflicts can cause ambiguity and problems.

Annotations

Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations don't change your code but will catch certain errors.

```
public class A {
  ...
  @Override
  public String toSpring() { //misspelled
    return "An object of type A";
  }
}
```

If a method annotated with `@Override` doesn't override anything, the compiler will issue

```
error: method does not override or implement a method
from a supertype
```

Annotations

Annotations are useful debugging tools. There are other annotations, and you can define custom annotations.

We will only use `@Override` in this course.

Widening primitive conversion

A type conversion is either *widening* or *narrowing*.

Widening primitive conversions include

```
int i = 100;
long li = i;
double d = i;
double d = (double) i;
float f = 12.3f;
double d2 = f;
```

Widening conversion has (almost) no loss of precision or information.
Because it is safe, widening conversion can happen implicitly.

Narrowing primitive conversion

Narrowing primitive conversions include

```
long li = 12312312312123123L;
//int i = li; //error
int i = (int) li;
double d = 0.3;
float f = (float) d;
int i2 = (int) d;
```

Information can be lost in narrowing conversion, if a big number doesn't fit into a smaller variable. Narrowing conversion doesn't happen implicitly.

Widening reference conversion

Converting a child class reference to a parent class reference is a widening conversion.

```
Srew s = new Screw(1,2,1);
Part p = s;
Part p = (Part) s;
```

Again, widening conversions can happen implicitly.

Narrowing reference conversion

Converting a parent class reference to a child class reference is a narrowing conversion.

```
Part p1 = new Part(1,2);
Part p2 = new Screw(1,2,1);
Srew s;
//s = p2; //compile-time error
s = (Screw) p2;
s = (Screw) p1; //runtime error
```

Again, narrowing conversion doesn't happen implicitly. Whether `p1` or `p2` does indeed refer to an object of `class Screw` is checked at runtime. (I.e., it's checked dynamically) If not, an error is issued.

A reference to a child class cannot refer to an object of a parent class.

instanceof

The relational operator `instanceof` checks the type of an object.

```
public static void main(String[] args) {
   A a1 = new A();
   A a2 = new B();
   System.out.println(a1 instanceof Object);//true
   System.out.println(a1 instanceof A); //true
   System.out.println(a1 instanceof B); //false
   System.out.println(a2 instanceof Object);//true
   System.out.println(a2 instanceof A); //true
   System.out.println(a2 instanceof B); //true
}
```

equals

All `classes` inherit the `equals` function from `Object`. By default it returns `true` if the two references refer to the same single object.

```
public class A {
  ...
  @Override
  public boolean equals(Object obj) {
    if (obj instanceof A) {
      //cast obj into type A
      A a = (A) obj;
      //check if this and a are equal
      ...
    } else
      return false;
  }
}
```

Override `equals` so that it returns `true` if the two references refer to objects of the same type with the same meaning.

Constructor chaining

A constructor is written as

```
accessModifier className(arguments) {
  explicitConstructorInvocation
  constructorBody
}
```

The explicit constructor invocation is something like `this(arguments)` or `super(arguments)`. It must be the first line of the constructor.

If a constructor does not explicitly invoke another constructor, it by default calls `super()`, the no-argument constructor of the superclass. `Object` has one constructor, which is a no-argument constructor.

Constructor chaining

So all constructors call another constructor until the no-argument constructor of `Object` is reached. As the sole exception, `Object`'s no-argument constructor doesn't call another constructor. This process is called *constructor chaining*.

This is the second use of the keyword `super`, explicitly invoking a superclass's constructor.

Constructor chaining

```
public class Complex {
  private final double real, imag;
  public Complex() {
    this(0., 0.);
  }
  public Complex(double real, double imag) {
    //super();
    this.real = real; this.imag = imag;
  }
}
```

When you run `new Complex()`,

1. `Complex()` is called
2. `Complex(double real, double imag)` is called, and
3. `Object()` is called.

Outline

abstract methods

An *abstract method* is declared without an implementation.

```
public abstract class Path {
  ...
  public abstract double getTime();
}
```

We want subclasses of `Path` to imeplement `getTime`, but there is no reasonable way to implement `getTime` for the superclass `Path`.

abstract classes

An *abstract class* may or may not include `abstract` methods.
An `abstract class` cannot be instantiated, but it can be inherited.

A non-`abstract class` (also called a *concrete* `class`) cannot have an `abstract` method.

In our example, the `class Path` must be an `abstract class`.
(What would it even mean to instantiate a non-specific `Path`?)

abstract classes

An `abstract class` lays out what its subclasses should do while only partially providing its implementation.

Say `class B` inherits an `abstract class A`. If `B` is non-`abstract`, it must implement the parent's `abstract` methods. If `B` is also `abstract`, the inherited `abstract` methods can be left `abstract`.

final methods

A `final` method cannot be overriden or hidden.

In a way `final` is the opposite of `abstract`. A `final` method cannot be overridden while an `abstract` method must be overridden.

Methods called from constructors should generally be declared `final`. If a constructor calls a non-`final` method, a subclass may override that method and bad things can happen.

final classes

A `final class` cannot be inherited.

As a general advice, you should design and document for inheritance or else prohibit it with `final`.

Again, `final` is the opposite of `abstract`, in a way. A `final class` cannot be inherited while an `abstract class` must be inherited.

Example: MusicFile

Imagine writing a music player application.

```
public abstract class MusicFile {
  ...
  public abstract void play();
  public abstract String songName();
}
```

Example: MusicFile

To play a music file, you decode the file and convert it into audio signals.

```
public class WAVFile extends MusicFile {
  ...
  @Override
  public void play() {
    //decode and play WAV file
    ...
  }
}
```

Example: MusicFile

The `play` method will be implemented differently for each music file type.

```
public class MP3File extends MusicFile {
  ...
  @Override
  public void play() {
    //decode and play MP3 file
    ...
  }
}
```

No music file is just a generic `MusicFile`, but all `MusicFiles` can `play`.

Example: MusicFile

When you write the music player program, the `abstract MusicFile class` is useful. It allows you to not worry about the specific kinds of music file you have.

```
public class Player {
  MusicFile[] song_arr;
  ...
  public void playList() {
    for (int i=0; i<song_arr.length; i++) {
      song_arr[i].play();
    }
  }
}
```

Different versions of `play` is called polymorphically.

Outline

Example

Imagine that some objects can be resold.

```
import java.util.Date;

public interface Resellable {
  //price of the object when sold at Date d
  double sellAt(Date d);
}
```

Example

A screw would likely sell at the same price at which it was bought.

```
public class Screw extends Part
                   implements Resellable {
  ...
  @Override
  public double sellAt(Date d) {
    return cost;
  }
}
```

Example

Perhaps leather degrades over time, its resell value depreciates.

```
public class SeatLeather extends Part
                         implements Resellable {
  ...
  @Override
  public double sellAt(Date d) {
    double price;
    //compute price while factoring depreciation
    ...
    return price;
  }
}
```

Example

```
import java.util.Date;

public class Car {
  private Part[] p_arr;
  ...
  public double sellValue() {
    Date date;
    ...
    double price = 0;
    for (Part p : p_arr)
      if (p instanceof Resellable)
        price += ((Resellable) p).sellAt(date);

    return price;
  }
}
```

What is an interface?

An `interface` is like an `abstract class`. It must be *implemented* (instead of inherited) to be instantiated.

You implement a `interface`, and you inherit a `class`.

View an `interface` as a *contract*. An `interface` specifies methods its implementation must provide.

Rules of interfaces

- All fields must be `public`, `static`, and `final`. (By default, they are, but you can redundantly specify these modifiers.) In other words, `interfaces` can only have constants as fields.
- `interfaces` are `abstract`. (By default, they are, but you can redundantly specify `abstract`.)
- `interfaces` do not have constructors.

Rules of interfaces

- There are exactly 3 types of methods for `interfaces`: `abstract`, `default`, and `static`.
- `abstract` is the default option. (You can redundantly specify `abstract`.)
- We'll talk about `default` methods for `interfaces` later.
- `static` methods for `interfaces` are rarely used.
- Methods must be `public`. (By default, they are, but you can redundantly specify `public`.)

abstract classes vs. interfaces

`interfaces` mostly specify `abstract` methods,
while `abstract classes` can have non-constant fields.

You can only inherit one `class`.
You can implement multiple `interfaces`.

Interfaces as function pointers

C++ and many other languages have function pointers.

```
int sum(int a, int b) {
  return a+b;
}
int max(int a, int b) {
  if (a>b) return a;
  else return b;
}
int reduce(int* arr, int n,
           int (*binOp)(int a, int b) ) {
  int ret = binOp(arr[0], arr[1]);
  for (int i=2; i<n; i++)
    ret = binOp(ret, arr[i]);
  return ret;
}
```

Interfaces as function pointers

C++ and many other languages have function pointers.

```
int main() {
  int n;
  int* arr;
  ...
  cout << reduce(arr, n, sum) << endl;
  cout << reduce(arr, n, max) << endl;
}
```

You use function pointers to make another function use the provided function. In this example, `reduce` uses `sum` and `max`.

Interfaces as function pointers

In Java, use `interfaces` instead of function pointers.

```
public interface BinOper {
  int op(int a, int b);
}
public class Adder implements BinOper {
  @Override
  public int op(int a, int b) {
    return a+b;
  }
}
public class Maxer implements BinOper {
  @Override
  public int op(int a, int b) {
    if (a>b) return a;
    else return b;
  }
}
```

Interfaces as function pointers

In Java, use `interfaces` instead of function pointers.

```
public static int reduce(int[] arr, BinOper b) {
  int ret = b.op(arr[0], arr[1]);
  for (int i=2; i<arr.length; i++)
    ret = b.op(ret, arr[i]);
  return ret;
}
```

Interfaces as function pointers

In Java, use `interfaces` instead of function pointers.

```
public static void main(String[] args) {
  int[] arr;
  ...
  int sum = reduce(arr, new Adder());
  int max = reduce(arr, new Maxer());
}
```

`Adder` and `Maxer` `Objects` don't have any fields but rather provide access to method `op`. Since `Adder` and `Maxer` `implements interface` `BinOper`, they must have the method `op`.

(`reduce` calls `op` polymorphically.)

Multiple inheritance

Inheriting/implementing from multiple `classes` and `interfaces` is often useful.

Java prohibits multiple inheritance from `classes`, as it multiple inheritance is very error prone. (C++ allows multiple inheritance.)

This is very rarely a limitation. Usually, you can do exactly what you want by implementing multiple `interfaces` without compromising any convenience or readability.

Example

Since countChar uses interface CharSequence and methods specified in CharSequence, it works with the classes CharBuffer, Segment, String, StringBuffer, and StringBuilder.

```
public int countChar(CharSequence str, char c) {
  int count = 0;
  for (int i=0; i<str.length(); i++)
    if (str.charAt(i)==c)
      count++;

  return count;
}
```

https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html

Outline

What is object-oriented programming?

Object-oriented programming (OOP) is a programming style centered around using objects.

A down-to-earth definition of objects: an object is an association of data and functions.

The four core tenets of OOP are

- abstraction,
- encapsulation,
- inheritence, and
- polymorphism.

Abstraction

Abstraction allows us to model a complex system with many small components using fewer larger but abstract components.

Sometimes abstract is easier.

When using the `class String` in a program, it's helpful to simply use methods like `charAt(int index)`, `length()`, or `toUpperCase()` without worrying about how they work. Even if you had write these methods yourself, it's easier to worry about that issue separately.

Abstraction is universal across all programming styles, but objects is a particularly elegant way to provide abstraction.

Encapsulation

An encapsulated `class` hides its fields from the user, and instead provides methods that provide indirect access to the fields.

Encapsulation can make your codebase safer and more readable.

Perfect encapsulation is often impossible. However, a limited degree of encapsulation is still useful.

Inheritance

One reason to use inheritance is to re-use code written for the parent class. For example, to add a functionality to an existing `class`, inherit it and write only the additional functionality you wish to add. You will do this in hw4.

Another reason to use inheritance is for polymorphism.

Polymorphism

Polymorphism is the ability to present the same interface for differing underlying data types.

When you refer to an object with a superclass reference and use overridden methods, you are using polymorphism.

Polymorphism is a very powerful tool and is one of the key strengths of inheritance.

Polymorphism is not exclusive to OOP, but OOP makes polymorphism clean.

When should you use inheritance and polymorphism?

Actually, less often then you might think.

- Some problems (e.g. the `Direction` example) are inherently object oriented, and inheritance/polymorphism is the right tool.
- We will soon see GUIs benefit from inheritance/polymorphism.
- HTML DOM is organized in an object oriented manner.
- Inheritance are useful in organizing a large and complex libraries like the Java API.
- However, inheritance is often misused and overused. Don't use inheritance just because you can.
- Inheritance/polymorphism is a tool. Use this tool when this tool makes the job easier.