Abstraction

November 8, 2019

https://en.wikipedia.org/wiki/Abstract_nonsense

Theories of Abstraction

Let's talk about abstraction.

Theories of Abstraction

Let's talk about abstraction.

- 

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
- Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.

Theories of Abstraction

Let's talk about abstraction.

- 
- Abstraction in visual art is about avoiding concrete subjects. It attempts to convey something (often an emotion) without appealing to sentiment.
- Oddly enough there's a bit of a stigma against it, despite the fact that music without lyrics has far less stigma.
- Abstraction in mathematics and computer science is about *generalization*. Take away the concrete details of certain objects and see how they are similar.

Why Abstract?

This can lead us to *classify* different objects into related groups.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
- But we will study abstraction from a less mathematical viewpoint.

Why Abstract?

This can lead us to *classify* different objects into related groups.

- Category theory has been influential in programming language theory because it is a powerful language for communicating abstractions in ways that are largely *constructive*.
- This is important to computing because the point of programs are to construct some form of answer.
- Although categorical abstractions are powerful, I will not discuss them much (and most of the really powerful abstractions go over my head).
- But we will study abstraction from a less mathematical viewpoint.
- I will show instances of *almost identical* code and how a programming language feature allows the two pieces of code to be generalized.

Buddy Functions

Let's consider the following functions in Java:

```
public boolean HasSmith(List<String> names) {
  for (String name : names) {
    if (name == "Smith")
      return true;
  }
  return false;
}

public  boolean HasBob(List<String> names) {
  for (String name : names) {
    if (name == "Bob")
      return true;
  }
  return false;
}
```

Buddy Functions

Let's consider the following functions in Java:

```
public boolean HasSmith(List<String> names) {
  for (String name : names) {
    if (name == "Smith")
      return true;
  }
  return false;
}

public  boolean HasBob(List<String> names) {
  for (String name : names) {
    if (name == "Bob")
      return true;
  }
  return false;
}
```

These look pretty similar, right?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean HasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean HasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean HasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?

Eliminate Redundancy!

So, let's eliminate the redundancy of searching for different names by abstracting out towards a definition that takes in a string parameter to search for. This generalizes writing functions to search for specific strings.

```
public boolean HasName(String searchName, List<String> names) {
  for (String name : names) {
    if (name == searchName)
      return true;
  }
  return false;
}
```

- Why only design a function that searches for names?
- Why not search for arbitrary items so long as they are comparable?
- Then searching for names becomes an instance of a more general problem that is solved.

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>> boolean HasItem(T searchItem, List
for (T item : items) {
  if (item.equals(searchItem))
    return true;
}
return false;
}
```

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>> boolean HasItem(T searchItem, List
for (T item : items) {
  if (item.equals(searchItem))
    return true;
}
return false;
}
```

- Alright, now we can search for arbitrary comparable items!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>> boolean HasItem(T searchItem, List
for (T item : items) {
  if (item.equals(searchItem))
    return true;
}
return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!

Generic Structure

So, let's use some Java *generics* to generalize our program.

```
public <T extends Comparable<T>> boolean HasItem(T searchItem, List
for (T item : items) {
  if (item.equals(searchItem))
    return true;
}
return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?

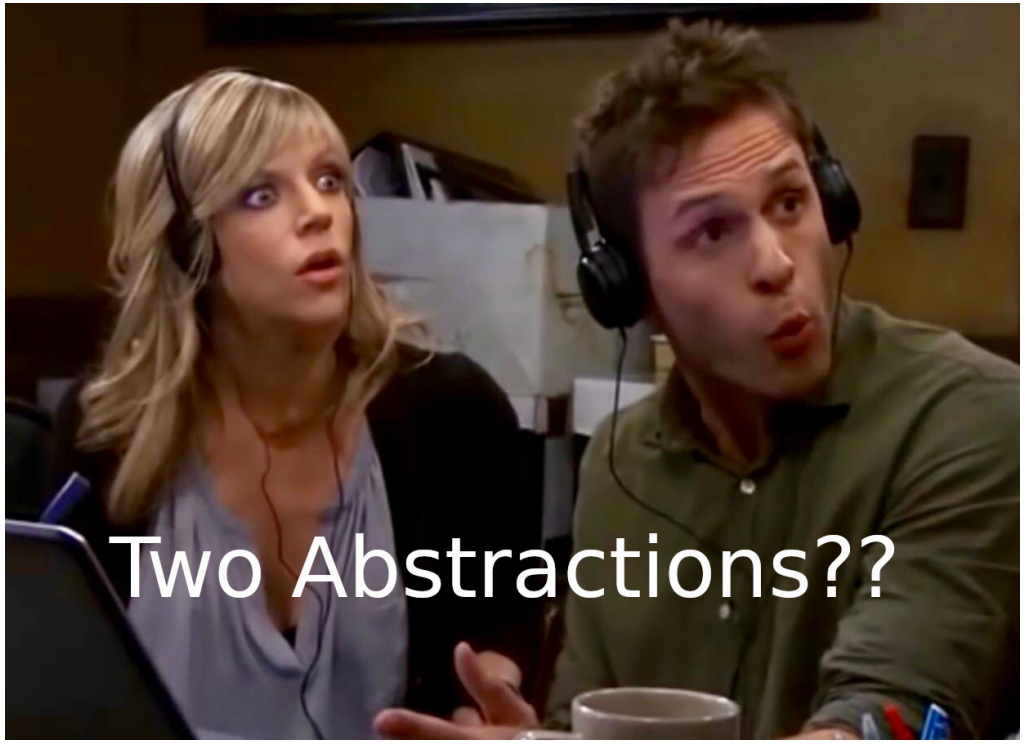
Generic Structure

So, let's use some Java *generics* to generalize our program.

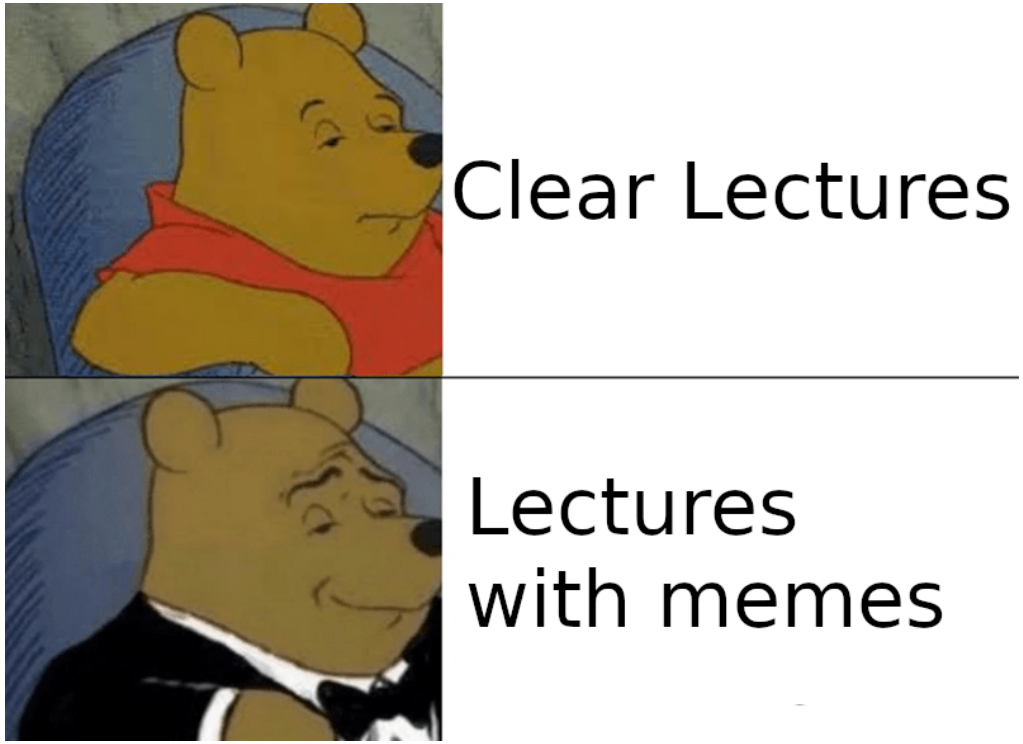
```
public <T extends Comparable<T>> boolean HasItem(T searchItem, List
for (T item : items) {
  if (item.equals(searchItem))
    return true;
}
return false;
}
```

- Alright, now we can search for arbitrary comparable items!
- This is nice and general!
- Can we generalize any more?
- We actually have 2 more abstractions that we can apply!

Two Abstractions??



I'm Sorry



Back on Track

Ok, let's get down to business.

Back on Track

Ok, let's get down to business.

- 

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:

```
Predicate<String> isSmith = str -> str == "Smith";
```

Back on Track

Ok, let's get down to business.

- 

- We can actually consider searching for a specific item via equality as the process of seeing if an arbitrary predicate returns true for an item in a list.
- So if we wanted to check if a string equaled smith we could write:
```
Predicate<String> isSmith = str -> str == "Smith";
```
- I could then pass this as the first argument to a funciton `TestItems` that I will now define.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>> boolean TestItems(Predicate<T> pre
for (T item : items) {
  if (pred.test(searchItem))
    return true;
}
return false;
}
```

Last Abstraction

Here is that function now:

```
    public <T extends Comparable<T>> boolean TestItems(Predicate<T> pre
    for (T item : items) {
      if (pred.test(searchItem))
        return true;
    }
    return false;
  }
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.

Last Abstraction

Here is that function now:

```
    public <T extends Comparable<T>> boolean TestItems(Predicate<T> pre
    for (T item : items) {
      if (pred.test(searchItem))
        return true;
    }
    return false;
  }
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

Last Abstraction

Here is that function now:

```
public <T extends Comparable<T>> boolean TestItems(Predicate<T> pre
for (T item : items) {
  if (pred.test(searchItem))
    return true;
}
return false;
}
```

- We can provide one last abstraction that applies predicates to every item in *any iterable collection*.
- For example, if I define iterators over dictionaries or trees we should still be able to apply a predicate to them.

```
public <T extends Comparable<T>> boolean TestItems(Predicate<T> pre
for (T item : items) {
  if (pred.test(searchItem))
    return true;
}
return false;
}
```