Contracts and Abstraction

- Casts
- Checking a Type
- Interfaces
Contracts

What is the contract for the `equals` method of `String`?

"hello".equals(...)

So far, we've pretended that it takes a `String` and produces a `boolean`

"hello".equals("bye") \to false

"hello".equals(8) \textit{contract mismatch}

The truth is somewhat more complex:

"hello".equals(new Posn(1, 2)) \to false
The Whole Truth

• The `equals` method takes an `Object` and returns a `boolean`.
• Every class extends `Object`.

```java
class Posn {
    double x;
    ...
}
```

is a shorthand for

```java
class Posn extends Object {
    double x;
    ...
}
```

• The `equals` method is defined in `Object`.
The Default Equals Method

class Object {
    ...
    
    boolean equals(Object o) {
        return o == this;
    }
}

where == is like eq? in Scheme
Using Object for Abstraction

In Scheme, we eventually wrote abstractions for lists:

```scheme
; A list-of-X is either
;   - empty
;   - (cons X list-of-X)
```

A precise translation to a Java-like notation:

```java
abstract class ListOf<X> { }
class EmptyListOf<X> { ... }
class ConsListOf<X> {
  <X> first;
  ListOf<X> rest;
  ...
}

new ConsListOf<String>("apple", ...)
```

But Java doesn't support this, yet
Using Object for Abstraction

In Scheme, we eventually wrote abstractions for lists:

```scheme
; A list-of-X is either
;   - empty
;   - (cons X list-of-X)
```

A usable translation to Java:

```java
abstract class List { }
class Empty { ... }
class Cons {
    Object first;
    List rest;
    ...
}

new Cons("apple", ...)
```
abstract class List {
    abstract boolean isMember(Object o);
}

class Empty extends List {
    Empty() { }
    boolean isMember(Object o) { return false; }
}

class Cons extends List {
    Object first;
    List rest;
    Cons(Object first, List rest) {
        this.first = first; this.rest = rest;
    }
    boolean isMember(Object o) { 
        return this.first.equals(o) || this.rest.isMember(o);
    }
}
Extracting Objects

• Implement the `List` method `nth`, which takes a number `n` and returns the first item in the list after skipping `n` items, or an empty list if no items are left.
Contracts and Abstraction

Casts

Checking a Type

Interfaces
Using Extracted Objects

\[
\text{new } \text{Cons}(\text{new } \text{Posn}(1, 2), \text{new } \text{Empty}()).\text{nth}(0) \\
\rightarrow \text{Posn}(x = 1, y = 2)
\]

\[
\text{new } \text{Cons}(\text{new } \text{Posn}(1, 2), \text{new } \text{Empty}()).\text{nth}(0).x \\
\text{contract error}
\]

The contract error occurs because \text{nth} promises merely to return an \text{Object}, not necessarily a \text{Posn}.

Java provides a way around this weakness in the contract system...
Casts

A **cast** is a dynamic request for an improved contract

General syntax:

```
(Class) expr
```

*The parentheses are required*

Examples:

```
(Posn)(new Cons(new Posn(1, 2), new Empty()).nth(0))
```

```java
Path escapePath(Person p) {
    Path lp = this.left.escapePath(p);
    if (lp.isOk())
        return new Left((Success)lp);
    ...
}
```
Using A Cast to implement equals

A problem with `Posn`:

\[
\text{new } \text{Posn}(1, 2).\text{equals(new Posn}(1, 2)) \rightarrow \text{false}
\]

To fix this, we need to override `equals`:

```java
class Posn {
    double x;
    double y;
    Posn(double x, double y) {
        this.x = x; this.y = y;
    }
    boolean equals(Object o) {
        return (this.x == ((Posn)o).x) && (this.y == ((Posn)o).y);
    }
}
```
- Contracts and Abstraction
- Casts
- Checking a Type
- Interfaces
Checking Types

A remaining problem:

```
"hello".equals(new Posn(1, 2)) \rightarrow false
new Posn(1, 2).equals("hello") \rightarrow cast failed
```

Our `equals` should only cast if the argument really is a `Posn`.

The `instanceof` operator tests whether a cast will succeed.

```java
boolean equals(Object o) {
    if (o instanceof Posn)
        return (this.x == ((Posn)o).x)
              && (this.y == ((Posn)o).y);
    else
        return false;
}
```
Using instanceof

The `instanceof` operator is only in Advanced Java because it's rarely the right way to implement something.

Example bad use:

```java
class Cons extends List {
    ...
    boolean isMember(Object o) {
        if (this.first.equals(o))
            return true;
        else if (this.rest instanceof Empty)
            return false;
        ...
    }
}
```
- Contracts and Abstraction
- Casts
- Checking a Type
- Interfaces
Named Doors

Suppose that we want to make the following improvements to our maze game:

- Some doors will have names
- We want to get all of the named places in a maze, including both escapes and named doors
- We'll need certain methods on named places, such as `isNice`
- We don't want to add named-place methods to all doors
- We refuse to use `instanceof`

```java
abstract class Door {
    ...
    abstract List places();
}
```
A NamedPlace Abstract Class

Like this?

```
NamedPlace

String name

boolean isNice()

Escape

boolean isNice()

NamedInto

boolean isNice()
```

**NamedPlace** can't be an abstract class, because **Escape** already extends **Door**, and **NamedInto** should extend **Into**

A class must extend exactly one class

However, **NamedPlace** can be an interface...
Interface

An interface is like an abstract class with no fields and all abstract methods

```java
interface NamedPlace {
    boolean isNice();
}
```

Instead of extending an interface, classes implement it

```java
class Escape extends Door implements NamedPlace {
    ...
    boolean isNice() { return true; }
}
```

```java
class NamedInto extends Into implements NamedPlace {
    ...
    boolean isNice() { return false; }
}
```
Door Hierarchy with Interfaces

- **NamedPlace**
  - ...

- **Door**
  - ...

- **Escape**
  - String name
  - ...

- **Into**
  - Room next
  - ...

- **NamedInto**
  - String name
  - ...

- **Short**
  - double height
  - ...

- **Locked**
  - String keyColor
  - ...

Single vs. Multiple, Implementation vs. Interface

A class must extend only one class
  ○ This is *single inheritance* of *implementation*

A class interface can implement any number of interfaces
  ○ This is *multiple inheritance* of *interface*