CSE 331

Subtypes

Kevin Zatloukal

Last Time on CSE 331

- Covered all the core theoretical material
  - covered on midterm and final
- Covered the core practical material
  - covered in HW8–9
- Remaining lectures will cover non-core topics
  - won’t be needed for HW
  - could be covered (small questions) on the final

Object-Oriented Programming

- We haven't done any OO this quarter
  - this week, we will see some reasons why!

- Plan for this week:
  - focus on topics that are good to know but not good for HW
    - usually, mistakes you want to avoid
  - every lecture will include one related to OO

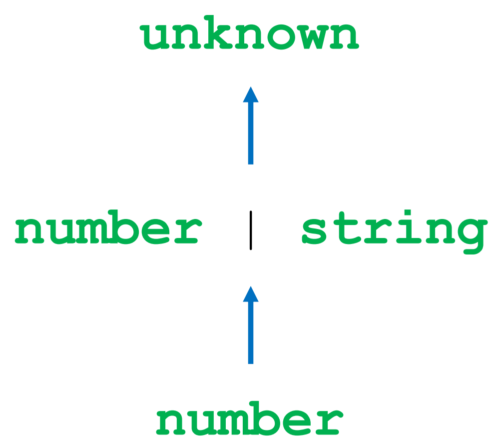
Subtypes

Subtypes of Concrete Types

- We initially defined types as sets
- In math, a **subtype** can be thought of as a **subset**
  - e.g., the even integers are a subtype of $\mathbb{Z}$
  - e.g., the numbers {1, 2, 3, 4, 5, 6} are a subtype of $\mathbb{Z}$
- Any even integer “is an” integer
  - “is a” is often (but not always) good intuition for subtypes

Subtypes of Concrete Types

- We initially defined types as sets
- In TypeScript, some subtypes are also subsets
  - `number` has a set of allowed values
  - it is a subtype of types that allow those values + more



Subtypes of Concrete Types

- We initially defined types as sets
- In TypeScript, some subtypes are also subsets
  - record types require certain fields but allow more
  - record type with a superset of the fields is a subtype

```
{name: string}
       ↑
{name: string, completed: boolean}
```

Subtyping Used by TypeScript

- **TypeScript uses subtyping in function calls**

```
function f(s: number | string): number { … }

const x: number = 3;
… f(x) …
```

  - **types are not the same** (`number` **vs** `number | string`)
  - **subtype can be passed where super-type is expected**

    any element of the subtype "is an" element of the super-type

- **Similar rules in Java**

Subtyping Used by TypeScript

- TypeScript uses subtyping in function calls

```
function f(n: number): number { … }

const x: number | string = f(3);
```

  - types are not the same (`number` vs `number | string`)
  - subtype can be returned where super-type is expected

    any element of the subtype "is an" element of the super-type

- Similar rules in Java

Subtyping Used by TypeScript

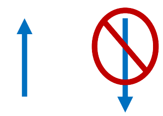
- TypeScript only sees the declared types
  - any other behavior is left to reasoning

- Example: invariants

```
// RI: 0 <= index < options.length
type OptionState = {
  options: string[],
  index: number
}
```

Subtyping Used by TypeScript

```
{options: string[], index: number}
```



```
OptionState
```

- `OptionState` **is a subtype of the bare record type**
  - **it is a record with those fields**
  - **but reverse is not true**

- **TypeScript will see these as the same**
  - **will let you pass the top where the bottom is expected**

    up to us to make sure this doesn't happen

Subtypes of Abstract Types

- Recall: ADTs are collections of functions
  - hide the concrete data
  - pass functions that operate on the data

    create, observe, mutate

- Subtypes are subsets does not work well here
  - set of all possible functions with … yuck

- Would be nice to find a cleaner approach

Subtypes Are Substitutable

- If B is a subtype of A, can send B where A is expected:

```
function f(s: A): void { … }
function g(): B { … }

const x: B = 3;
f(x);                    // okay

const y: A = g();  // okay
```

  – okay to “substitute” a B where an A is expected

Subtypes Are Substitutable

- Subtypes are **substitutable** for supertype
  - this is the "Liskov substitution principle"
  - due to Barbra Liskov
- For ADTs, we use this as our definition of subtypes
  - (for concrete types, subsets are usually easier)

Subtypes of Abstract Types

- When is ADT B substitutable for A?

- Must satisfy two conditions:

  1. B must provide all the methods of A

     If A has a method "f", then B must have a method called "f"

  2. B's corresponding method must…

     must accept all the inputs that A's does

     must also promise everything in A's postcondition

     I.e., B must have the same or a **stronger spec**

Review: Strengthening a Specification

```
interface A {
  f(x: number): number

  // @requires x >= 0
  g(x: number): number
}
```

- Stronger specs allow more (or same) inputs
  - allowed argument types are supersets

```
interface B extends A {
  f(x: number | string): number
}
```

  - fewer requirements on arguments

```
interface C extends A {
  g(x: number): number     // x can be negative
}
```

Review: Strengthening a Specification

```
interface A {
  f(x: number): number

  // @requires x >= 0
  g(x: number): number
}
```

- Stronger specs promise more (or same) outputs
  - more specific return type (or thrown type)

```
interface D extends A {
  f(x: number): 0 | 1 | 2 | 3
}
```

Review: Strengthening a Specification

```
interface A {
  f(x: number): number

  // @requires x >= 0
  g(x: number): number
}
```

- Stronger specs promise more (or same) outputs
  - more specific return type (or thrown type)
  - more facts included in @returns and @effects

```
interface E extends A {
  // @requires x >= 0
  // @returns an even integer
  g(x: number): number
}
```

  - fewer objects listed in @modifies

Example: Rectangle and Square

- Is Square a subtype of Rectangle?
  - math intuition says yes
  - a square "is a" rectangle
- Let's check this with substitutability…

Example: Immutable Rectangle and Square

```
interface Rectangle {
  getWidth(): number,
  getHeight(): number
}

// A rectangle with width = height
interface Square extends Rectangle {
  getSideLength(): number
}
```

extra invariant
on abstract state

Yes

- Is `Square` substitutable for `Rectangle`?
  - allows the same inputs (none)
  - makes the same promises about outputs (numbers)
  - adds another promise: both methods return same number

Example: Mutable Rectangle and Square

```
interface Rectangle {
  getWidth(): number,
  getHeight(): number
  resize(width: number, height: number): void
}

// A rectangle with width = height
interface Square extends Rectangle {
  // @requires width = height
  resize(width: number, height: number): void
}
```

- Is `Square` substitutable for `Rectangle`? No!
  - allows fewer inputs to resize!

Example: Mutable Rectangle and Square

- None of these work:

```
// @requires width = height
resize(width: number, height: number): void

// @throws Error if width != height
resize(width: number, height: number): void

// Sets height = width also
resize(width: number): void
```

incomparable specs

- Mutation sometimes makes subtyping impossible
  – yet another reason to avoid it

Subclasses

Subclasses

- Java subclassing is a means of sharing code
  - subclass gets parent fields & methods (unless overridden)

```
class Product {
  private String name;
  private int price;
  public String getName() {return name; }
  public int getPrice() { return price; }
}

class SaleProduct extends Product {
  private float discount;
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Subclasses

- **Subclassing does not guaranty subtyping relationship**

```
class Product {
  public int getPrice() { ... }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return getPrice() < p.getPrice();
  }
}

class WackyProduct extends Product {
  // @returns some boolean value
  public boolean isCheaperThan(Product p) {
    return false;
  }
}
```

Legal Java, but not a subtype

Subclasses

- Java subclassing is a means of sharing code
  - subclass gets parent fields & methods (unless overridden)
- Does not guarantee subtyping
  - up to you to check that method specs are stronger
- Java treats it as a subtype
  - will let you pass subclasses where superclass is expected
- Subclassing is a surprisingly dangerous feature
  - that's not the only reason…

Subclasses

- Subclassing is a surprisingly dangerous feature

- Subclassing tends to break modularity
  - creates tight coupling between super- and sub-class
  - often see the “fragile base class” problem

    changes to super class often break subclasses

- Let’s see some Java examples…

Example 1: Tight Coupling

```
class Product {
  private int price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return getPrice() < p.getPrice();
  }
}

class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

– looks okay so far…

Example 1: Tight Coupling

```
class Product {
  private String price;
  public int getPrice() { return price; }

  // @returns true iff obj's price < p's price
  public boolean isCheaperThan(Product p) {
    return this.price < p.price;
  }
}

class SaleProduct extends Product {
  public int getPrice() {
    return (1 - discount) * super.getPrice();
  }
}
```

Made it faster by eliminating a method call!

What's wrong?

Oops! Broke the subclass

Example 2: Tight Coupling

```
class InstrumentedHashSet extends HashSet<Integer> {
  private static int count = 0;

  public boolean add(Integer e) {
    count += 1;
    return super.add(e);
  }

  public boolean addAll(Collection<Integer> c) {
    count += c.size();
    return super.addAll(c);
  }

  public int getCount() { return count; }
}
```

– what could possibly go wrong?

Example 2: Tight Coupling

```
InstrumentedHashSet S = new InstrumentedHashSet();
System.out.println(S.getCount());  // 0
S.addAll(Arrays.asList(1, 2));
System.out.println(S.getCount());  // 4?!?
```

– **what does this print?**

- **What is printed depends on `HashSet`'s `addAll`:**
  - **if it calls `add`, then this prints 4**
  - **if it does not call `add`, then this prints 2**

- **Also possible to be dependent on *order* of calls**

Example 3: Tight Coupling

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    addToLists(job.getName(), job.getTime());
    total += job.getTime();
  }

  protected addToLists(String name, int time) {
    names.add(name);
    times.add(time);
  }
}
```

Example 3: Tight Coupling

```
// Makes sure no task is too large compared to rest
class BalancedWorkList extends WorkList {
  protected addToLists(String name, int time) {
    if (times.size() <= 3 || 2*time < total)
      super.addToLists(name, time);   // okay
    } else {
      throw new ImbalancedWorkException(name, time);
    }
  }
}
```

– prevents item from being added if too big
– (also: this subclass is not a subtype!)

Example 3: Tight Coupling

```
class WorkList {
  // RI: len(names) = len(times) and total = sum(times)
  protected ArrayList<String> names;
  protected ArrayList<Integer> times;
  protected int total;

  public addWork(Job job) {
    int time = job.getTime();  // just one call
    total += time;
    addToLists(job.getName(), time);
  }
}
```

RI not true in method call

– reordering the updates breaks the subclass!
– subclass is using `total` that includes the new job

Example 3: Tight Coupling

- RI can be false in calls to non-public methods
  – only needs to hold at end of the public method

- Requires extra care to get it right
  – method is tightly coupled with the ones that call it
  – needs to know what is true in those methods
    not enough to just know the RI

- Hard for multiple people to communicate this clearly
  – can be okay when it's all your code
  – very error prone when methods are written by others

Subclassing Creates Tight Coupling

- Creates tight coupling between super- and sub-class
  - direct field access can break subclass
  - subclass dependent on which methods call each other
  - subclass dependent no *order* of method class
  - subclass can be called when RI is false

- Often see the “fragile base class” problem

- Subclassing is a surprisingly dangerous feature!
  - up to you to verify subclass method specs are stronger
  - up to you to prevent tight coupling

Subclassing is Best Avoided

- Java advice: either design for subclassing or prohibit it
  - from Josh Bloch, author of (much of) the Java libraries
- We haven't used subclassing in TypeScript
  - didn't even describe how to do it!
    - we've just used classes as a quick way to create records
  - these problems are the main reason why we avoided it
- Subclassing is not necessary anyway
  - we have other ways to share code