CSE331 SU Final Simple Summary

Matt XU

Subtyping Stuffs

Subtypes are substitutable

Subtypes are ***substitutable*** for supertypes

- Instances of subtype won't surprise client by failing to satisfy the supertype's specification
- Instances of subtype won't surprise client by having more expectations than the supertype's specification

This follows the "Principle of Least Surprise"

We say that B is a ***true subtype*** of A if B has a stronger specification than A

- This is ***not*** the same as a ***Java* subtype**
- Java subtypes that are not true subtypes are *confusing* and *dangerous*
  - But unfortunately common poor-design ☹

Subtyping vs. subclassing

Substitution (subtype) — a specification notion

- B is a subtype of A iff an object of B can masquerade as an object of A in any context
- About satisfiability (behavior of a B is a subset of A's spec)

Inheritance (subclass) — an implementation notion

- Factor out repeated code
- To create a new class, write only the differences

Java purposely merges these notions for classes:

- Every subclass is a Java subtype
  - But not necessarily a true subtype

Cheat Sheet

- B is a true subtype of A. How do I code this up?
  - Use java subclassing! (B extends A)
- B is not a true subtype of A, but shares a lot with A. How do I code this up?
  - It's tempting to use java subclassing when B is not a true subtype of A (Square/Rectangle)
    - avoid it, since you might run into issues like the square/rectangle issue
  - But I don't want to duplicate all the code in A. Duplication is evil.
    - you're right! try Composition. (B has a A)
- B is a true subtype of A, but has an entirely different implementation. I don't want to inherit anything, but Java needs to know they're the same type for polymorphism to work. How do I code this up?
  - A and B should implement the same interface.

Cheat Sheet

- B is a true subtype of A, but A is an existing class that I can't modify and it's not subclass-ready (Hashtable/InstrumentedHashTable)
  - Composition will be helpful here too! (B has a A)
  - And, if possible, have B implement the same interface as A, for polymorphism.
- D is a true subtype of A and of T. Java only has single inheritance. How do I code up this relationship?
  - Use interfaces. D can implement interface A and interface T. Or extend one as a class and implement the other as an interface.

Assertions & Exceptions

What to do when something goes wrong

Fail early, fail friendly

Goal 1: *Give information about the problem*

- To the programmer – a good error message is key!
- To the client code: via exception or return-value or …

Goal 2: *Prevent harm*

Abort: inform a human

- Perform cleanup actions, log the error, etc.

Re-try:

- Problem might be transient

Skip a subcomputation:

- Permit rest of program to continue

Fix the problem?

- *Usually* infeasible to repair from an unexpected state

Avoiding errors

A precondition prohibits misuse of your code
- Adding a precondition weakens the spec

This ducks the problem of errors-will-happen
- Mistakes in your own code
- Misuse of your code by others

Removing a precondition requires specifying more behavior
- Often a good thing, but there are tradeoffs
- Strengthens the spec
- Example: specify that an exception is thrown

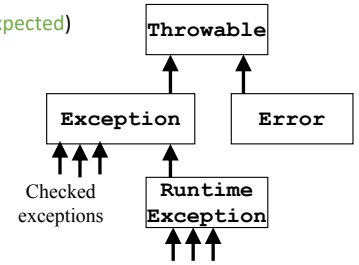
Recall your experience

Sometimes recalling what you have encountered this quarter and how you solved those program failures can help you strengthen your perception of these things.

Java's checked/unchecked distinction

Checked exceptions (*style*: for *special cases*)
- Callee: *Must* declare in signature (else type error)
- Client: Must either catch or declare (else type error)
  - Even if *you* can prove it will never happen at run time, the type system does not "believe you"
- There is guaranteed to be a dynamically enclosing catch

Unchecked exceptions (*style*: for never-expected)
- Library: No need to declare
- Client: No *need* to catch
- Subclasses of `RuntimeException` and `Error`



Why catch exceptions locally?

Failure to catch exceptions usually violates modularity
- Call chain: A → IntegerSet.insert → IntegerList.insert
- IntegerList.insert throws some exception
  - Implementer of IntegerSet.insert knows how list is being used
  - Implementer of A may not even know that IntegerList exists

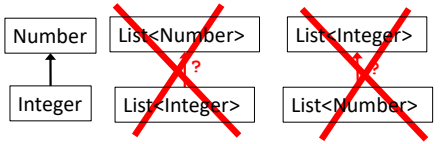
Method on the stack may think that it is handling an exception raised by a different call

Better alternative: catch it and throw again
- "chaining" or "translation"
- Restate in a level of abstraction that the client can understand
- Do this even if the exception is better handled up a level
- Makes it clear to reader of code that it was not an omission

Generics

`List<Number>` and `List<Integer>`

```
interface List<T> {
  boolean add(T elt);
  T get(int index);
}
```



So type `List<Number>` has:

```
boolean add(Number elt);
Number get(int index);
```

So type `List<Integer>` has:

```
boolean add(Integer elt);
Integer get(int index);
```

- Subtype needs stronger spec than super
- Stronger method spec has:
  - weaker precondition
  - stronger postcondition

Java subtyping is *invariant* with respect to generics

- Neither `List<Number>` nor `List<Integer>` subtype of other
- Not covariant and not contravariant

Generic types and subtyping

- `List<Integer>` and `List<Number>` are not subtype-related
- Generic types can have subtyping relationships
- Example: If `HeftyBag` extends `Bag`, then
  - `HeftyBag<Integer>` is a subtype of `Bag<Integer>`
  - `HeftyBag<Number>` is a subtype of `Bag<Number>`
  - `HeftyBag<String>` is a subtype of `Bag<String>`
  - …

Reasoning about wildcard types

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? extends Integer> lei;
```

First, which of these is legal?

~~`lei = new ArrayList<Object>();`~~
~~`lei = new ArrayList<Number>();`~~
`lei = new ArrayList<Integer>();`
`lei = new ArrayList<PositiveInteger>();`
`lei = new ArrayList<NegativeInteger>();`

Which of these is legal?

~~`lei.add(o);`~~
~~`lei.add(n);`~~
~~`lei.add(i);`~~
~~`lei.add(p);`~~
`lei.add(null);`
`o = lei.get(0);`
`n = lei.get(0);`
`i = lei.get(0);`
~~`p = lei.get(0);`~~

Reasoning about wildcard types

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? super Integer> lsi;
```

First, which of these is legal?

```
lsi = new ArrayList<Object>;
lsi = new ArrayList<Number>;
lsi = new ArrayList<Integer>;
lsi = new ArrayList<PositiveInteger>;
lsi = new ArrayList<NegativeInteger>;
```

Which of these is legal?

```
lsi.add(o);
lsi.add(n);
lsi.add(i);
lsi.add(p);
lsi.add(null);
o = lsi.get(0);
n = lsi.get(0);
i = lsi.get(0);
p = lsi.get(0);
```

Type erasure

All generic types become type **`Object`** once compiled

- Big reason: backward compatibility with ancient byte code
- So, at run-time, all generic instantiations have the same type

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

Cannot use **`instanceof`** to discover a type parameter

```
Collection<String> cs = new ArrayList<String>();
  if (cs instanceof Collection<String>) { // illegal
  ...
}
```

Assertions & Exceptions

- Don’t hide errors
- Be systematic
- Recall your stories

Callbacks

The callback design pattern

Going farther: use a callback to *invert the dependency*

`TimeToStretch` creates a **`Timer`**, and passes in a reference to *itself* so the **`Timer`** can *call it back*

- This is a *callback* – a method call from a module to a client that it notifies about some condition

The callback *inverts a dependency*

- Inverted dependency: **`TimeToStretch`** depends on **`Timer`** (not vice versa)
  - Less obvious coding style, but more "natural" dependency
- Side benefit: **`Main`** does not depend on **`Timer`**

***Read the slides for observers

Design Patterns

Look at our section slides and handouts.

Also for the System Development stuffs	That’s it