Exam 2 Review

Exam 2
- Exam 2 on Tuesday, November 3rd
- Closed book, closed computer, closed phone
- You are allowed two “cheat” pages
  - Either a standard 8.5x11-sized sheet filled on both sides or two standard sheets filled on one side each
  - Type or write by hand (which I recommend)

ADTs
- Reasoning about Abstract Data Type (ADT)
  - Rep invariants, AFs, prove that rep invariant holds by induction

How to Design Your Code
- The hard way: Start hacking. When something doesn’t work, hack some more
- The easier way: Plan carefully
  - Write specs, rep invariants, abstraction functions
  - Write tests (first!), reason about code, refactor
  - Less apparent progress at first, but faster completion times, better product, less frustration

How to Verify Your Code
- The hard way: hacking, make up some inputs
- An easier way: systematic testing
  - Black-box testing techniques (more next time)
  - High white-box coverage (more next time)
  - JUnit framework
- Also: reasoning, complementary to testing
  - We’ll prove that rep invariant is preserved
  - We’ll write informal proofs!

Ways to Make New Objects
- a = constructor
- b = a.producer
- a’ = a.mutator
- a” = a.observer
- c = b.producer
- b’ = b.mutator
- b” = b.observer
- ... Infinitely many objects but limited number of operations!
Induction

- Proving facts about infinitely many objects
- Base step
  - Prove rep invariant holds on exit of constructor
- Inductive step
  - Assume rep invariant holds on entry of method
  - Then prove that rep invariant holds on exit
- Intuitively: there is no way to make an object, for which the rep invariant does not hold
- Remember, our proofs are informal

Reasoning About Rep Invariant

- Inductive step must consider all possible changes to the rep
  - Including representation exposure
  - If the proof does not account for representation exposure, then it is invalid!
  - That’s why exposure of immutable rep is OK!

Topics

- Exceptions
  - Preconditions vs. exceptions, throwing and catching down the call stack, exceptions vs. special values, checked vs. unchecked exceptions

Preconditions vs. Exceptions

- In certain cases, preconditions are a valid choice
  - When checking is expensive. E.g., binarySearch
  - In private methods, usually used in local context
  - Whenever possible, remove preconditions from public methods and specify behavior
    - Usually, this entails throwing an Exception
    - Stronger spec, easier to use by client

Throwing and Catching

- Java maintains a call stack of methods that are currently executing
- When an exception is thrown, control transfers to the nearest method with a matching catch block
  - If none found, top-level handler
- Exceptions allow for non-local error handling
  - A method far down the call stack can handle a deep error!

Informing the Client of a Problem

- Special value
  - null – Map.get(x)
  - -1 – List.indexOf(x)
  - NaN – sqrt of negative number
- Problems with using special value
  - Hard to distinguish from real values
  - Error-prone: programmer forgets to check result? The value is illegal and will cause problems later
  - Ugly
- Exceptions are generally a better way to inform of a problem!
Two Distinct Uses of Exceptions

- Failures
  - Unexpected by your code
  - Usually unrecoverable. If condition is left unchecked, exception propagates down the stack

- Special results
  - Expected by your code
  - Unknowable for the client of your code
  - Always check and handle locally. Take special action and continue computing

Java Exceptions: Checked vs. Unchecked Exceptions

- Checked exceptions. For special results
  - Library: must declare in signature
  - Client: must either catch or declare in signature
  - It is guaranteed there is a dynamically enclosing catch

- Unchecked exceptions. For failures
  - Library: no need to declare
  - Client: no need to catch
  - RuntimeException and Error
  - Always check and handle locally. Take special action and continue computing

Topics

- Testing
  - Black box heuristics: equivalence partitioning, boundary value analysis, white box heuristics: control-flow graph (CFG), statement coverage, branch coverage, def-use coverage.

Testing Strategies

- Test case: specifies
  - Inputs + pre-test state of the software
  - Expected result (outputs and post-test state)

- Black box testing:
  - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
  - Choose inputs without looking at the code

- White box (clear box, glass box) testing:
  - We use knowledge of the code of the program (roughly, we write tests to "cover" internal paths)
  - Choose inputs with knowledge of implementation

Equivalence Partitioning

- Partition the input and/or output domains into equivalence classes
- Write tests with inputs from different equivalence classes
- Write tests that will produce outputs from different equivalence classes

Boundary Value Analysis

- Choose test inputs at the edges of the input equivalence classes
- Choose test inputs that produce outputs at the edges of output equivalence classes
- Other boundary cases
  - Arithmetic: zero, overflow
  - Objects: null, circular list, aliasing
Control-flow Graph (CFG)

- Assignment: $x = y + z$ => node in CFG: $\text{node: } x = y + z$
- If-then-else:
  ```plaintext
  if (b) S1 else S2 =>
  CFG for S1
  CFG for S2
  end-if
  ```

Coverage

- Statement coverage: Write a test suite that covers all statements, or in other words, all nodes in the CFG
- Branch coverage: Write a test suite that covers all branch edges at predicate nodes
  - The True and False edge at if-then-else
  - The two branch edges corresponding to the condition of a loop
  - All alternatives in a SWITCH statement

White Box Testing: Dataflow-based Testing

- A definition (def) of $x$ is $x$ at the left-hand-side
  - E.g., $x = y + z$, $x = x + 1$, $x = \text{foo}(y)$
- A use of $x$ is when $x$ is at the right-hand side
  - E.g., $z = x + y$, $x = x + y$, $x > y$, $z = \text{foo}(x)$
- A def-use pair of $x$ is a pair of nodes, $k$ and $n$ in the CFG, s.t. $k$ is a def of $x$, $n$ is a use of $x$, and there is a path from $k$ to $n$ free of definition of $x$

White Box Testing: Dataflow-based Testing

- Dataflow-based testing targets: write tests that cover paths between def-use pairs
- Intuition:
  - If code computed a wrong value at a def of $x$, the more uses of this def of $x$ we "cover", the higher the possibility that we’ll expose the error
  - If code had erroneous use of $x$, the more def-use pairs we "cover", the higher the possibility that we’ll expose the error at the use of $x$

A Buggy gcd

```c
// requires a, b > 0
static int gcd(int a, int b) {
    int x = a;
    int y = b;
    while (x != y) {
        if (x > y) {
            x = x - 2*y;
        } else {
            y = y - x;
        }
    }
    return x;
}
```

Let’s test with $\text{gcd}(15, 6)$ and $\text{gcd}(4, 8)$.
What’s the statement coverage? Branch?
1. x=a; y=b;
2. x!=y  False
3. x>y  True
4. x=x-2y;  y=y-x; 
5. True
6. False
7. res=x;

Def-use coverage targets: cover paths connecting def-use pairs

Def-use Coverage Targets

- The All-defs coverage target: for every def x, cover at least one path (free of definition of x), to at least one use x
- The All-uses coverage target: for every def-use pair of x, cover at least one path (free of definition of x) from the def x to the use x
- The All-du-paths coverage target: for every def-use pair of x, cover every path (free of definition of x) from the def x to the use x

Subtype Polymorphism

- Subtype polymorphism – the ability to use a subclass where a superclass is expected
  - Thus, dynamic method binding
    - class A { void m() { ... } }
    - class B extends A { void m() { ... } }
    - class C extends A { void m() { ... } }
    - Client: A a; a.m(); Call a.m() can bind to any of A.m, B.m, or C.m at runtime!
  - Subtype polymorphism is a language feature
  - Java subtype: B extends A or B implements I
  - A Java subtype is not necessarily a true subtype!

Subtypes are Substitutable

- Subtypes are substitutable for supertypes
  - Instances of subtype won’t surprise client by expecting more than the supertype
  - Instances of subtypes won’t surprise client by failing to satisfy supertype postcondition
- B is a true subtype (or “behavioral” subtype) of A if B has stronger specification than A
  - Not the same as Java subtype!
  - Java subtypes that are not true subtypes are confusing and dangerous

Benefits of Subtype Polymorphism

- “Science” of software design teaches Design Patterns
- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism
Liskov Substitution Principle (LSP)
- Due to Barbara Liskov, Turing Award 2008
- LSP: A subclass B should be substitutable for its superclass A. i.e., B is a true subtype of A.
- To ensure that B is substitutable:
  - B does not remove methods from A
  - For each B.m that "replaces" A.m, B.m's specification is stronger than A.m's specification
    - Example: Client A a; ... a.m(int x, int y); Call a.m can bind to B's m. B.m should not surprise client

Function Subtyping
- In programming languages function subtyping deals with substitutability of functions
  - Question: under what conditions on the parameter and return types A,B,C and D, is function A f(B) substitutable for C f(D)?
  - Reasons at the level of the type signature
    - Rule: A f(B) is a function subtype of C f(D) if A is a subtype of C and B is a supertype of D
    - Guarantees substitutability!

Type Signature of Substituting Method is Stronger
- Method parameters (inputs):
  - Parameter types of A.m may be replaced by supertypes in subclass B.m. "contravariance"
    - E.g., A.m(String p) and B.m(Object p)
  - B.m places no extra requirements on the client!
    - E.g., client: A a; ... a.m(q). Client knows to provide q a String. Thus, client code will work fine with B.m(Object p), which asks for less: an Object, and clearly, every String is an Object.
  - Java does not allow change of parameter types in an overriding method. More on Java overriding shortly

Type Signature of Substituting Method is Stronger
- Method returns (results):
  - Return type of A.m may be replaced by subtype in subclass B.m. "covariance"
    - E.g., Object A.m() and String B.m()
  - B.m does not violate expectations of the client!
    - E.g., client: A a; ... Object o = a.m(). Client expects an Object. Thus, String will work fine
  - No new exceptions. Existing exceptions can be replaced by subtypes
  - Java does allow a subtype return type in an overriding method!

Reasoning about Specs
- Function subtyping reasons with type signatures
  - Remember, type signature is a specification
    - Precondition: requires arguments of given type
    - Postcondition: promises result of given type
  - Compiler checks function subtyping
  - Specifications add reasoning about behavior and effects
    - Precondition: stated by requires clause
    - Postcondition: stated by modifies, effects, returns and throws clauses

Reason about Specs
- "Behavioral" subtyping generalizes function subtyping
  - B.m is a true subtype (behavioral subtype) of A.m
    - B.m has weaker precondition than A.m
      - Generalizes "B.m's parameter is a supertype of A.m's parameter" premise of function subtyping rule
    - Covariance
    - B.m has stronger postcondition than A.m
      - Generalizes "B.m's return is a subtype of A.m's return" Covariance
  - These 2 conditions guarantee B.m's spec is stronger than A.m's spec, and B.m is substitutable for A.m
Java Subtypes

- Java types are defined by classes, interfaces, primitives
- Java subtyping stems from declarations
  - B extends A
  - B implements A
- In a Java subtype, a “substituting” method is an overriding method
  - Has same parameter types
  - Has compatible (same or subtype) return type
  - Has no additional declared exceptions

Java Subtypes: Overloading vs. Overriding

- A method family contains multiple implementations of same name + parameter types subsignature (no return type)
- The method family is determined at compile time based on static types of receiver and parameters
  - E.g., family put(Object key, Object value)
  - or family put(String key, String value)
- Which implementation from the method family runs, is determined at runtime based on type of the receiver

Exercise

class Y extends X { ... }  // A a = new B();
class A {  // Which m is called?
x m(Object o) { ... }  // x = a.m(o);
}
class B extends A {  // x = a.m(o);
    X m(Z z) { ... }
}
class C extends B {  // x = a.m(o);
    Y m(Z z) { ... }
}

Exercise

class Y extends X { ... }  // A a = new B();
class W extends Z { ... }  // W w = new W();
class A {  // Which m is called?
    X m(Z z) { ... }  // x = a.m(w);
}
class B extends A {  // x = a.m(w);
    X m(W w) { ... }
}
class C extends B {  // x = b.m(w);
    Y m(W w) { ... }
}

Topics

- Equality
  - Properties of equality, reference vs. value
equality, equality and inheritance, equals and
hashCode, equality and mutation

Equality: == and equals()

- In Java, == tests for reference equality. This is the strongest form of equality
- Usually we need a weaker form of equality, value equality
- In our Point example, we want x to be "equal" to y because
  the x and y objects hold the same value
- Need to override Object.equals

Fall 15 CSCI 2600, A Milanova
Properties of Equality

- Equality is an equivalence relation
  - Reflexive: \( a \text{.equals}(a) \)
  - Symmetric: \( a \text{.equals}(b) \iff b \text{.equals}(a) \)
  - Transitive: \( a \text{.equals}(b) \land b \text{.equals}(c) \iff a \text{.equals}(c) \)

Equality and Inheritance

- Let \( B \) extend \( A \)
  - "Natural" definition of \( B \text{.equals}(Object) \) may lose symmetry
  - "Fix" may render \( \text{equals}() \) non-transitive

One can avoid these issues by defining equality for exact classes (has pitfalls too)

```java
if (!o.getClass().equals(getClass()))
    return false;
```

equals and hashCode

- \( \text{hashCode} \) computes an index for the object (to be used in hashtables)
- Javadoc for \( Object\text{.hashCode}() \):
  - "Returns a hash code value of the object. This method is supported for the benefit of hashtables such as those provided by HashMap."
  - Self-consistent: \( o\text{.hashCode()} == o\text{.hashCode()} \)
  - ... as long as \( o \) does not change between the calls
  - Consistent with \( \text{equals}() \) method: \( a\text{.equals}(b) \implies a\text{.hashCode()} == b\text{.hashCode()} \)

Equality, mutation and time

- If two objects are equal now, will they always be equal?
  - In mathematics, the answer is "yes"
  - In Java, the answer is "you chose"
  - The Object spec does not specify this
  - For immutable objects
    - Abstract value never changes, equality is eternal
  - For mutable objects
    - We can either compare abstract values now, or
    - be eternal (can’t have both since value can change)

Equality and Mutation

- Client may violate rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating elements after insertion

```java
Set<Date> s = new HashSet<Date>();
Date d1 = new Date(0);
Date d2 = new Date(1);
s.add(d1);
s.add(d2);
d2.setTime(0); // mutation after d2 already in the Set!
for (Date d : s) { System.out.println(d); }
```

Topics

- Parametric Polymorphism
  - Declaring and instantiating generics, Bounded types, Wildcards, Type erasure, Arrays
Polymorphism

- **Subtype polymorphism**
  - Code can use a subclass $B$ where a superclass $A$ is expected
  - Standard in object-oriented languages

- **Parametric polymorphism**
  - Code takes a type as a parameter
  - **Implicit** parametric polymorphism
  - **Explicit** parametric polymorphism

Using Java Generics

```java
List<String> names = new ArrayList<String>();
names.add("Ana");
names.add("Katarina");
String s = names.get(0); // what happens here?
Point p = names.get(0); // what happens here?
Point p = (Point) names.get(0); // what happens?
```

### Using Java Generics

```java
List<AType> list = new ArrayList<AType>();
```

AType is the type argument. We instantiated generic (templated) class `ArrayList` with concrete type argument `AType`.

## Defining a Generic Class

```java
class MySet<T> {
    // rep invariant: non-null, contains no duplicates
    List<T> theRep;
    T lastLookedUp;
}
```

### Bounded Type Parameters

- **<Type extends SuperType>**
  - An upper bound, type argument can be `SuperType` or any of its subtypes

- **<Type super SubType>**
  - A lower bound, type argument can be `SubType` or any of its supertypes

### Exercise

Given this hierarchy with $X$, $Y$ and $Z$:

```java
class A<T extends X> { ... }?
class A<T extends Z> { ... }?
class A<T super Z> { ... }?
class A<T super X> { ... }?
```

More Bounded Type Examples

```java
<T extends Comparable<T>>
    T max(Collection<T> c);
```

```java
<T> void copy(List<T2 super T> dst, List<T3 extends T> src);
```

(actualy, must use wildcard ? --- more on this later:
```java
<T> void copy(List<T> super T> dst, List<T> extends T> src); )
```

```java
<T extends Comparable<T2 super T>>
    void sort(List<T2 super T> list)
```
What is the Subtyping Relationship Between List<Number> and List<Integer>

- Java subtyping is invariant with respect to generics: if A ≠ B, then C<A> has no subtyping relationship with C<B>

Thus, List<Number> and List<Integer> are unrelated through subtyping!

Invariance is Restrictive (Because it Disallows Subtyping!)

- Java solution: wildcards

```java
interface Set<E> {
    // Ads all elements in c to this set
    // if they are not already present.
    void addAll(Set<E> c);
    void addAll(Collection<E> c);
    <T extends E> void addAll(Collection<T> c);
}
```

Not good either. Can’t have Set<Number> s; List<Integer> l; s.addAll(l);

This is because of invariance: List<Integer> is a subtype of Collection<Integer> but Collection<Integer> is not a subtype of Collection<Number>!

Java Wildcards

- A wildcard is essentially an anonymous type variable
  - Use ? if you’d use a type variable exactly once
  - ? appears at the instantiation site of the generic (also called use site)
  - As opposed to declaration site (also called definition site: where type parameter is declared)

Purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping

Legal Operations on Wildcards

Object o;
Number n;
Integer i;
PositiveInteger p;
List<? extends Integer> lei;

```java
lei = new ArrayList<Object>();
lei = new ArrayList<Number>();
lei = new ArrayList<Integer>();
lei = new ArrayList<PositiveInteger>();
lei = new ArrayList<NegativeInteger>();
```
Arrays and Subtyping

- `Integer` is subtype of `Number`.
- Is `Integer[]` a subtype of `Number[]`?
- Use our subtyping rules to find out (Just like with List<Integer> and List<Number>)
- Again, the answer is NO!
- Different answer in Java: in Java `Integer[]` is a Java subtype `Number[]`!
- The Java subtype is not a true subtype!
- Known as “problem with Java’s covariant arrays”.

---

Integer[] is a Java subtype of Number[]

```java
Number n;
Number[] na; // happens here?
Integer i;
Integer[] ia;
na[0] = n;
na[1] = i;
n = na[0];
i = na[1]; // what happens?
s = na[0]; // what happens?
na = ia; // OK!
na[2] = d;
na[1] = i;
in = ia[0]; // what happens?
a[i] = na[1]; // what happens?
a[0] = n; // what happens?
a[1] = i;
n = ia[0];
i = ia[1]; // what happens here?
```