THANKS!

- Professor Barb Cutler for the Homework Server!
- Mentors: Brandon, Will and Leopold
- TAs: Pranshu and Anshul

Final Exam

- Mon, Dec 21 3-6pm in DCC 318
- Final exam is cumulative
- Closed notes, closed laptops/phones
- 4 “cheat” pages, either four single-sided or two double-sided standard 8.5x11-sized sheets
- Type or write by hand (which I recommend)

How to study

- Review today’s slides
- Review Exams 1 and 2
- Back tests, posted on Announcements page
- No mentor and no TA office hours next week
- I’ll have office hours Friday, Dec 18, 10-2pm and Monday 21, 10-2pm

Grades

- CHECK YOUR GRADES IN THE LMS!
  - HW7 and HW8 feedback and grades in the Homework server. Let me know if you have questions
  - HW9 will be graded by Friday, Dec 18
  - You will have all your grades except for the final exam grade into the LMS by Friday, Dec 18

PoS is about writing correct and maintainable software

- Specifications
- Polymorphism, abstraction and modularity
- Design patterns
- Refactoring
- Reasoning about code
- Testing
- Software process
- Tools - Java, Eclipse, Subversion, Junit, EclEmma
- Principles are far more important than tools!
PoS is about writing correct and maintainable software

- Building correct software is hard!
  - Lots of dependencies
  - Lots of “moving parts”
- Software engineering is primarily about mitigating and managing complexity
  - Specifications, abstraction, design patterns, refactoring, reasoning about code (invariants “fix” one part, thus fewer “moving parts” to worry about), testing
  - All of these mitigate complexity

Outline

Quiz questions in reverse chronological order

Review of topics in chronological order

Quiz 10 Questions *

```java
JButton b = new JButton("Ouch");
b.addActionListener(new ActionListener() {
    void actionPerformed(ActionEvent e) {
        doSomething();
    }
});
```

Closest design pattern:

- a) Adapter
- b) Observer
- c) Interpreter
- d) Anonymous Class

Quiz 9 Questions

A library class accepts and returns values in English units but you need metric units. What design pattern allows us to reuse this library class?

- a) Adapter
- b) Decorator
- c) Proxy
- d) Delegation

To repaint in response to button click, we must call paintComponent(...) from doSomething()?

- a) True
- b) False

Quiz 10 Questions *

```java
JButton b = new JButton("Ouch");
b.addActionListener(new ActionListener() {
    void actionPerformed(ActionEvent e) {
        doSomething();
    }
});
```

The call b.addActionListener(...) is a callback?

- a) True
- b) False
Quiz 9 Questions

A design pattern used to enhance functionality of an object is

- a) Adapter
- b) Proxy
- c) Delegation

Design pattern(s) that traverse a hierarchical structure

- a) Procedural
- b) Observer
- c) Interpreter
- d) Visitor

Fall 15 CSCI 2600, A Milanova 13

Quiz 9 Questions

In the Java Collections library class AbstractCollection implements methods contains and equals as these methods work identically across all concrete Collections. contains and equals call iterator() which returns an Iterator over the collection. However, AbstractCollection.iterator() is abstract, thus deferring creation of the Iterator to the concrete subclass. What design pattern(s) is used here?

- a) Factory Method
- b) Factory Object
- c) Template Method
- d) Inheritance

Fall 15 CSCI 2600, A Milanova 14

A Question I got rid of…

What’s wrong with Willy’s implementation of interned boxed Integers?

```java
class Int {
    private int value;
    private Int (int value) { this.value = value; }
    private static Map<Int,Int> cache = new HashMap<>();
    public static Int valueOf(int value) {
        Int tmp = new Int(value);
        if (cache.containsKey(tmp))
            return cache.get(tmp);
        else {
            cache.put(tmp,tmp);
            return tmp;
        }
    }
}
```

Fall 15 CSCI 2600, A Milanova 15

Quiz 8 Questions

Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;
pei.add(pi);

a) Is legal
b) Is not legal

Fall 15 CSCI 2600, A Milanova 16

Quiz 8 Questions

Number n;
Integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;
i = pei.poll();

a) Is legal
b) Is not legal

Fall 15 CSCI 2600, A Milanova 17
Quiz 8 Questions

Number n;
integer i;
PositiveInteger pi;
NegativeInteger ni;
PriorityQueue<? extends Integer> pei;
PriorityQueue<? super Integer> psi;

pei = psi;
a) Is legal
b) Is not legal

Quiz 7 Questions

If there exist non-null references x, y and z such that x.equals(y) is false, y.equals(z) is true and x.equals(z) is true, then equals is not transitive.
a) True
b) False

The consistency property of hashCode requires that for every non-null x and y such that x.equals(y) is false, x.hashCode() != y.hashCode().
a) True
b) False

Quiz 7 Questions

int f(int y) {
    int s = 0;
    int x = 0;
    while (x<y) {
        x = x+3;
        y = y+2;
        if (x+y<10) {
            s = s+x+y;
        } else {
            s = s+x-y;
        } // end-if
    } // end-while
    return s;
}

Draw the CFG for the function.

Find an argument a such that f(a) achieves 100% statement coverage.

Is it possible to cover def-use pair (6:s=s+x-y,5:s=s+x+y)?
a) Yes
b) No

Specification tests is just another name for black-box tests.
a) True
b) False

Quiz 8 Questions

PriorityQueue<E> constructor:
PriorityQueue(Comparator<? super E> comparator)

PriorityQueue<Integer> pqi;
Comparator<Integer> cn;
Comparator<Integer> ci;
Comparator<PositiveInteger> cp;

Circle the legal instantiations:
- pqi = new PriorityQueue<Integer>(cn);
- pqi = new PriorityQueue<Integer>(ci);
- pqi = new PriorityQueue<Integer>(cp);

Topics

- Reasoning about code
- Specifications
- ADTs, rep invariants and abs. functions
- Testing
- Subtyping vs. subclassing
- Equality
- Design patterns and refactoring
- Usability, Software process, Requirements
Topics

- Reasoning about code
  - Forward and backward reasoning, logical conditions, Hoare triples, weakest precondition, rules for assignment, sequence, if-then-else, loops, loop invariants, decrementing functions

Forward Reasoning

- Forward reasoning simulates the execution of the code. Introduces facts as it goes along
  
  E.g., \( \{ x = 1 \} \)
  
  \[
  \begin{align*}
  y &= 2 \times x \\
  \{ x = 1 \ \text{AND} \ y = 2 \}
  \end{align*}
  \]
  
  \[
  \begin{align*}
  z &= x + y \\
  \{ x = 1 \ \text{AND} \ y = 2 \ \text{AND} \ z = 3 \}
  \end{align*}
  \]

  - Collects all facts, often those facts are irrelevant to the goal

Backward Reasoning

- Backward reasoning “goes backwards”. Starting from a postcondition, finds the weakest precondition that ensures the given postcondition
  
  E.g., \( \{ 2y < y+1 \} \) // Simplify into \( y < 1 \)
  
  \[
  \begin{align*}
  z &= y + 1 \\
  \{ 2y < z \}
  \end{align*}
  \]
  
  \[
  \begin{align*}
  x &= 2 \times y \\
  \{ x < z \}
  \end{align*}
  \]

  - More focused and more useful

Condition Strength

- “P is stronger than Q” means “P implies Q”
- “P is stronger than Q” means “P guarantees more than Q”
  
  - E.g., \( x > 0 \) is stronger than \( x > -1 \)
  
  - Fewer values satisfy P than Q
    - E.g., fewer values satisfy \( x > 0 \) than \( x > -1 \)

  - Stronger means more specific
  - Weaker means more general

Exercise. Condition Strength

- Which one is stronger?

  \[
  \begin{align*}
  x &> -10 \quad \text{or} \quad x > 0 \\
  x &> 0 \ \&\& \ y = 0 \ \text{or} \ x > 0 \ \| \ y = 0 \\
  0 &\leq x \leq 10 \ \text{or} \ 5 \leq x \leq 11 \\
  y &\equiv 2 \ (\text{mod} \ 4) \ \text{or} \ y \text{ is even} \\
  y &\equiv 1 \ (\text{mod} \ 3) \ \text{or} \ y \text{ is odd} \\
  x &= 10 \ \text{or} \ x \text{ is even}
  \end{align*}
  \]

Hoare Triples

- A Hoare Triple: \( \{ P \} \ \text{code} \{ Q \} \)
  
  - P and Q are logical conditions (statements) about program values, and code is program code (in our case, Java code)
  
  - \( \{ P \} \ \text{code} \{ Q \} \) means “if P is true and we execute code, then Q is true afterwards”

  - \( \{ P \} \ \text{code} \{ Q \} \) is a logical formula, just like \( 0 \leq \text{index} \)
Exercises. Hoare Triples

{ x>0 } x++ { x>1 } is true
{ x>0 } x++ { x>-1 } is true
{ x≥0 } x++ { x>1 } is false. Why?

{x>0} x++ {x>0} is ??
{x<0} x++ {x<0} is ??
{x=a} if (x < 0) x=-x { x = | a | } is ??
{x=y} x=x+3 {x=y} is ??

Exercise

- Let P => Q => R
  (P is stronger than Q and Q is stronger than R)
- Let S => T => U
- Let { Q } code { T }
- Which of the following are true:
  1. { P } code { T }
  2. { R } code { T }
  3. { P } code { U }
  4. { P } code { S }

Rules for Backward Reasoning: Assignment

// precondition: ??
x = expression
// postcondition: Q

Rule: the weakest precondition = Q, with all occurrences of x in Q replaced by expression
More formally:
\( wp("x=expression;",Q) = Q \) with all occurrences of x replaced by expression

Rules for Backward Reasoning: Sequence

// precondition: ??
S1; // statement
S2; // another statement
// postcondition: Q

Work backwards: precondition is \( wp("S1;S2;",". Q) = wp("S1;",". wp("S2;",".Q)) \)

Example:
// precondition: ??
x = 0;
y = x+1;
// postcondition: y>0
// precondition for x = 0; same as
x = 0;
// postcondition for y = x+1;
y = x+1;
// postcondition y>0

Exercise

Compute the weakest precondition:

if (x < 0) {
  y = -x;
} else {
  y = x;
} 
\{ y = |x| \}
Exercise

Find the strongest postcondition:
\{ p^2 + q^2 = r \}
\begin{align*}
r &= r/p \\
q &= q*q/p
\end{align*}

Reasoning About Loops by Induction

1. Partial correctness
   - Guess and prove loop invariant using computation induction
   - Loop exit condition and loop invariant must imply the desired postcondition

2. Termination
   - (Intuitively) Establish “decrementing function” D.
     Each iteration decrements D, D = 0 and loop invariant, imply loop exit condition

Example: Reasoning About Loops

Precondition: x >= 0;
i = x;
z = 0;
while (i != 0) {
    z = z+1;
i = i-1;
}
Postcondition: x = z;
Need to prove:
1. x = z holds after the loop (partial correctness)
2. Loop terminates (termination)

Reasoning About Loops

Loop invariant Inv must be such that
1) P \Rightarrow Inv // Inv holds before loop. Base case
2) \{ Inv \land b \} S \{ Inv \} // Assuming Inv held after k-th iteration and execution took a (k+1)-th iteration, then Inv holds after (k+1)-th iteration. Induction
3) (Inv \land \neg b) \Rightarrow Q // The exit condition \neg b and loop invariant Inv must imply postcondition

Decrementing function D must be such that
1) D decreases every time we go through the loop
2) D = 0 and Inv must imply loop exit condition \neg b

Exercise

Find the weakest precondition

\begin{align*}
y &= x + 4; \\
\text{if } (x > 0) \{ \\
    y &= x*x - 1; \\
}\text{else} \{
    y &= y + x;
}\end{align*}

Precondition: y >= 0;
i = y;
n = 1;
while (i != 0) {
    n = n*x;
i = i-1;
}
Postcondition: n = x^n;

Prove partial correctness and termination
Specifications

A specification consists of a precondition and a postcondition

- Precondition: conditions that hold before method executes
- Postcondition: conditions that hold after method finished execution (if precondition held!)

Benefits of Specifications

- Document method behavior
- Imagine if you had to read the code of the Java libraries to figure what they do!
- An abstraction – abstracts away unnecessary detail
- Promotes modularity
- Enables reasoning about correctness
  - Through testing and/or verification

Example Specification

```
Precondition: len ≥ 0 ∧ arr.length = len
double sum(int[] arr, int len) {
    double sum = 0.0;
    int i = 0;
    while (i < len) {
        sum = sum + arr[i];
        i = i+1;
    }
    return sum;
}
```

Postcondition: returns arr[0]+...+arr[arr.length-1]

PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
  - requires: clause spells out constraints on client
- The postcondition
  - modifies: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - throws: lists possible exceptions
  - effects: describes final state of modified objects
  - returns: describes return value
Exercise

```java
static List<Integer> listAdd(List<Integer> lst1,
          List<Integer> lst2)
  requires: lst1 is non-null and lst2 is non-null
  modifies:
  effects:
  returns:
```

```java
static List<Integer> listAdd(List<Integer> lst1,
          List<Integer> lst2) {
  List<Integer> res = new ArrayList<Integer>();
  for (int i = 0; i < lst1.size(); i++)
    res.add(lst1.get(i) + lst2.get(i));
  return res;
}
```

Exercise

```java
static void listAdd2(List<Integer> lst1,
          List<Integer> lst2)
  requires: lst1 non-null and lst2 non-null
  modifies:
  effects:
  returns:
```

```java
static void listAdd(List<Integer> lst1,
          List<Integer> lst2) {
  for (int i = 0; i < lst1.size(); i++) {
    lst1.set(i, lst1.get(i)+lst2.get(i));
  }
}
```

Specification Strength

- "A is stronger than B" means
  - For every implementation I
    - "I satisfies A" implies "I satisfies B"
    - The opposite is not necessarily true
  - For every client C
    - "C meets the obligations of B" implies "C meets the obligations of A"
    - The opposite is not necessarily true
- Principle of substitutability:
  - A stronger spec can always be substituted for a weaker one

Specification Strength and Modularity

Client => Library L1

Library L2

L2 must be stronger than L1

Spec strength, Substitutability and Modularity

**Bad!** Y surprises the client!
The principle of substitutability tells us that if the specification of Y, `foo`, is stronger than the specification of X, `foo`, then it will be ok to use Y. `foo`!

Strengthening and Weakening Specification

- Strengthen a specification
  - Require less of client: fewer conditions in requires clause AND/OR
  - Promise more to client: effects, modifies, returns
    - Effects/modifies affect fewer objects
- Weaken a specification
  - Require more of client: add conditions to requires AND/OR
  - Promise less to client: effects, modifies, returns clauses are weaker, thus easier to satisfy in code.
Comparing Specifications by Logical Formulas

- Specification A is a logical formula: \( P_A \Rightarrow Q_A \)
  (meaning, precondition of A implies postcondition of A)
- Spec A is stronger than Spec B if and only if for each implementation \( I \), \((I \text{ satisfies } A) \Rightarrow (I \text{ satisfies } B)\)
  which is equivalent to \( A \Rightarrow B \)
- \( A \Rightarrow B \) means \( (P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) \)

Recall from FoCS and/or Intro to Logic: \( p \Rightarrow q \equiv !p \lor q \)

Comparing Specifications by Logical Formulas

\[ (P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) = \]
\[ \neg(P_A \Rightarrow Q_A) \lor (P_B \Rightarrow Q_B) = \]
\[ (P_A \lor \neg Q_A) \lor (P_B \Rightarrow Q_B) = \]
\[ ((P_B \lor Q_B) \land (P_A \lor \neg Q_A)) \Rightarrow (P_B \lor Q_B) \land (P_A \lor \neg Q_A) \]

[Due to commutativity of \( \lor \)]

\[ (P_B \Rightarrow (Q_B \lor P_A)) \land ((P_B \land Q_A) \Rightarrow Q_B) \]
Translation: A is stronger than B if and only if \( P_B \) implies \( Q_B \) or \( P_A \) AND \( Q_B \), together with \( P_B \) implies \( Q_B \).

Comparing Specifications by Logical Formulas

\[ \text{if and only if} \]
\[ (P_B \Rightarrow (Q_B \lor P_A)) \land ((P_B \land Q_A) \Rightarrow Q_B) \]

Sometimes we use the simpler test:
Spec A is stronger than Spec B if
\( P_B \Rightarrow P_A \) and \( Q_A \Rightarrow Q_B \)

Example:
\[ \text{int find(int[]} \, a, \text{int value}) \]

- Specification B:
  \[ \text{requires: } a \text{ is non-null and value occurs in } a \]
  \[ \text{returns: } i \text{ such that } a[i] = \text{value} \]

- Specification A:
  \[ \text{requires: } a \text{ is non-null} \]
  \[ \text{returns: } i \text{ such that } a[i] = \text{value} \text{ if value occurs in } a \]

Clearly, \( P_B = P_A \) (\( P_A \) includes \( P_B \) and one more condition)
Also, \( P_B \land Q_A \Rightarrow Q_B \). \( P_B \) says that “\( \text{value occurs in } a \)” and \( Q_A \) says “\( \text{value occurs in } a \) => returns i such that \( a[i]=\text{value} \)”. Thus, “returns i such that \( a[i]=\text{value} \)” which is exactly \( Q_B \), holds.

Exercise: Order by Strength

Spec A: \text{requires: a non-negative int argument}
\text{returns: an int in [1..10]}

Spec B: \text{requires: int argument}
\text{returns: an int in [2..5]}

Spec C: \text{requires: true}
\text{returns: an int in [2..5]}

Spec D: \text{requires: an int in [1..10]}
\text{returns: an int in [1..20]}

Function Subtyping

- Method inputs:
  - Parameter types of \( B.m \) may be replaced by supertypes in subclass \( A.m \). “contravariance”
    \[ \text{e.g., } B.m(\text{Integer } p) \text{ and } A.m(\text{Number } p) \]
  - This places no extra requirements on the client!
    \[ \text{e.g., client: } b : \ b.m(\text{q}) \]. Client knows to provide \( q \) a Integer or a subtype of Integer. Thus, client code
    \text{will work fine with } A.m(\text{Number } p), \text{ which asks for less: an Number or a subtype of Number}

- Java does not allow change of parameter types in an overriding method. More on Java overriding shortly.
Function Subtyping

- Method results:
  - Return type of `B.m` may be replaced by subtypes in subclass `A.m`. "covariance"
  - E.g., `Number B.m()` and `Integer A.m()`
  - This does not violate expectations of the client!
    - E.g., client: `B b; Number n = b.m()`. Client expects a Number. Thus, Integer will work fine
  - No new exceptions. Existing exceptions can be replaced by subtypes
  - Java does allow a subtype return type in an overriding method!

Exercise

- `A`'s `m`: `X m(X y, String s)`
- Let `Z` us a subtype of `Y`, `Y` is subtype of `X`, which `m` is function subtype of `A`'s `m`?
- `B`'s `m`:
  - `Y m(Object y, Object s)`
  - `Z m(Y y, String s)`

How to Use Wildcards

- Use `<E extends T>` when you `get` (read) values from a producer (is return)
- Use `<E super T>` when you `add` (write) values into a consumer (is parameter)
- E.g.:
  - `<T> void copy(List<? super T> dst, List<? extends T> src)`
  - PECS: Producer Extends. Consumer Super
  - Use neither, just `<T>`, if both `add` and `get`

Using Wildcards

```
class HashSet<E> implements Set<E> {
  void addAll(Collection<? extends E> c) {
    // What does this give us about c?
    // i.e., what can code assume about c?
    // What operations can code invoke on c?
  }
}
```

Legal Operations on Wildcards

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

First, which of these is legal?
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
lei.add(null);
lei = new ArrayList<Object>();
lei = new ArrayList<Number>();
lei = new ArrayList<Integer>();
lei = new ArrayList<PositiveInteger>();
lei = new ArrayList<NegativeInteger>();
```

Using Wildcards

```
class PriorityQueue<E> extends AbstractQueue<E> {
  PriorityQueue(int capacity, Comparator<? super E> c) {
    // What does this give us about c?
    // i.e., what can code assume about c?
    // What operations can code invoke on c?
  }
}
```
Legal Operations on Wildcards

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>();

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);
lsi = new ArrayList<PositiveInteger>();

Topics

ADTs, representation invariants and abstraction functions
  Benefits of ADT methodology, Specifying ADTs
  Rep invariant, abstraction function, representation exposure, checkRep, properties of abstraction function, benevolent side effects, proving rep invariants

An ADT Is a Set of Operations

Operations operate on data representation
ADT abstracts from organization to meaning of data
ADT abstracts from structure to use
Data representation does not matter!

class Point {
  float x, y;
}
class Point {
  float r, theta;
}

Instead, think of a type as a set of operations: create, x(), y(), r(), theta().
Force clients to call operations to access data

Specifying an ADT

immutable   mutable
class TypeName   class TypeName
  1. overview     1. overview
  2. abstract fields     2. abstract fields
  3. creators
  4. observers
  5. producers (rare!)
  6. mutators

Connecting Implementation to Specification

Representation invariant: Object \rightarrow boolean
  Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  Defines the set of valid values

Abstraction function: Object \rightarrow abstract value
  What the data structure really means
  E.g., array [2, 3, -1] represents \(-x^2 + 3x + 2\)
  How the data structure is to be interpreted
Representation Exposure

- Suppose we add this method to IntSet:
  ```java
  public List<Integer> getElements() {
      return data;
  }
  ```
- Now client has direct access to the rep `data`, can modify rep and break rep invariant.
- **Representation exposure** is external access to the rep. **AVOID**!
- Better: make a copy on the way out; make a copy on the way in.

Checking Rep Invariant

- Always check if rep invariant holds when debugging.
- Leave checks anyway, if they are inexpensive.
- Checking rep invariant of IntSet:
  ```java
  private void checkRep() {
      for (int i=0; i<data.size; i++)
          if (data.indexOf(data.elementAt(i)) != i)
              throw RuntimeException("duplicates");
  }
  ```

Abstraction Function: mapping rep to abstract value

- Abstraction function: Object $\rightarrow$ abstract value
  - I.e., the object’s rep maps to abstract value
    - IntSet e.g.: `list [2, 3, 1] $\rightarrow$ {1, 2, 3}`
    - Many objects map to the same abstract value
      - IntSet e.g.: `[2, 3, 1] $\rightarrow$ {1, 2, 3}` and `[3, 1, 2] $\rightarrow$ {1, 2, 3} and `[1, 2, 3] $\rightarrow$ {1, 2, 3}`
  - Not a function in the opposite direction
    - One abstract value maps to many objects

Correctness

- Abstraction function allows us to reason about correctness of the implementation

IntSet Example

- Creating concrete object: Establish rep invariant
  - `[1,2,3]`
  - `AF:
    - `[2,1,1,2,3]`
    - `Concrete remove(1)`
  - After every operations: Maintain rep invariant
  - `Equ.: (2,3)`
  - `AF:`
  - `[2,2,3]`

Proving rep invariants by 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 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.
Exercise: Willy's IntStack
Prove rep invariant holds

```java
class IntStack {
    // Rep invariant: |theRep| = size
    // and theRep.keySet = {i | 1 ≤ i ≤ size}
    private IntMap theRep = new IntMap();
    private int size = 0;

    public void push(int val) {
        size = size + 1;
        theRep.put(size, val);
    }

    public int pop() {
        int val = theRep.remove(size);
        size = size - 1;
        return val;
    }
}
```

Exercise: Willy's IntStack

Base case
- Prove rep invariant holds on exit of constructor

Inductive step
- Prove that if rep invariant holds on entry of method, it holds on exit of method
  - push
  - Pop
- For brevity, ignore popping an empty stack

What if Willy added this method:

```java
public IntMap getMap() {
    return theRep;
}
```

Does the proof still hold?

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
- E.g., spec of sqrt(double x):
  - returns: square root of x if x ≥ 0
  - throws: IllegalArgumentException if x < 0
- Partition the input domain
  - E.g., test x < 0, test x = 0, test x ≥ 0
- Partition the output domain too
  - E.g., test x < 1, x = 1, x > 1 (something interesting happens at 1)

Boundary Value Analysis

Choose test inputs at the edges of the input equivalence classes
- Sqrt example: test with 0,
- 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 $x = y + z \Rightarrow$ node in CFG: $x = y + z$
- If-then-else
  - $if \ (b) \ S1 \ else \ S2 \Rightarrow$
  - $CFG \ for \ S1$
  - $CFG \ for \ S2$
- Loop
  - $while \ (b) \ S \Rightarrow$
- $CFG \ for \ S$

**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
- **Def-use coverage**

**Exercise**

- Draw the CFG for
  // requires: positive integers $a, b$
  ```java
  static int gcd(int a, int b) {
    while (a != b) {
      if (a > b) {
        a = a - 2b;
      } else {
        b = b - a;
      }
    }
    return a;
  }
  ```
  What is %branch coverage for $gcd(15, 6)$?

**Topics**

- **Subtyping vs. subclassing**
  - Subtype polymorphism, true subtypes and the LSP, specification strength and function subtyping, Java subtypes (overriding and overloading)

**Subtype Polymorphism**

- **Subtype polymorphism** – the ability to use a subclass where a superclass is expected
  - Thus, dynamic method binding
    ```java
    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 --- essential object-oriented language feature
- Java subtype: $B$ extends $A$ or $B$ implements $I$
- A Java subtype is not necessarily a true subtype.
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!

What is True Subtyping?

- Also called behavioral subtyping
- A true subtype is not only a Java subtype but a "behavioral subtype"
- B is subtype of A means every B is an A
- B shall "behave" as an A
  - B shall require no more than A
  - B shall promise at least as much as A
- In other words, B will do fine where an A is expected

Subtypes are Substitutable

- Subtypes are substitutable for supertypes
  - Instances of subtypes won’t surprise client by requiring more than the supertype’s specification
  - Instances of subtypes won’t surprise client by failing to satisfy supertype specification
- 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 substitutable are confusing and dangerous

Liskov Substitution Principle (LSP)

- Due to Barbara Liskov, Turing Award 2008
- LSP: A subclass B of A should be substitutable for A, i.e., B should be a true subtype of A
- Reasoning about substitutability of B for A
  - B should not remove methods from A
  - For each B.m, which "substitutes" A.m, B.m's specification is stronger than A.m's specification
    - client: A a; ... a.m(int x, int y);
    - call a.m can bind to B's m and B's m should not surprise client

Overloading vs. Overriding

- A method family contains multiple implementations of same name + parameter types (but not return type!)
- Which method family is determined at compile time based on compile-time types
  - 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 the type of the receiver

Exercise

```java
class VarExp extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
}
```

Why not move `void accept(Visitor v)` up into superclass `BooleanExp`?

```java
class Evaluate implements Visitor {
    // state, needed to
    // evaluate
    void visit(VarExp e) {
        // evaluate Var exp
        void visit(Constant e) {
            // evaluate And exp
            // visit for all exps
        }
    }
    ...
}
```

class PrettyPrint implements Visitor {
    ...
}
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

Properties of Equality
- Equality is an equivalence relation
  - Reflexive: a.equals(a)
  - Symmetric: a.equals(b) ⇔ b.equals(a)
  - Transitive: a.equals(b) ∧ b.equals(c) ⇔ a.equals(c)

Equality and Inheritance
- Let B extend A
- "Natural" definition of B.equals is not symmetric
- Fix renders equals non transitive
- One can avoid these issues by allowing equality for exact classes:
  if (!o.getClass().equals(getClass()))
  return false;

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

Mutation can violate rep invariant of a Set container (rep invariant: there are no duplicates in set) by mutating after insertion

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); }

Exercise:
Remember Duration

class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o); // override
    public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Duration d)

Exercise:
Remember Duration

class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Duration d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Object o)
// At runtime: Duration.equals(Object o)

Exercise:
Remember Duration

class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Duration d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Object o)
// At runtime: Duration.equals(Object o)

Exercise:
Remember Duration

class Y extends X { … }
class A {
    X m(Object o) { … }
}
class B extends A {
    X m(Z z) { … }
}
class C extends B {
    Y m(Z z) { … }
}
A a = new B();
Object o = new Object();
X x = a.m(o);
A a = new C();
Object o = new Z();
X x = a.m(o);
Exercise

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

A a = new B();
W w = new W();
// Which m is called?
X x = a.m(w);

B b = new C();
W w = new W();
// Which m is called?
X x = b.m(w);

Topics

- Design Patterns
  - Creational patterns: Factory method, Factory class, Prototype, Singleton, Interning
  - Structural patterns:
    - Wrappers: Adapter, Decorator, Proxy
    - Composite
    - Façade
  - Behavioral patterns:
    - Interpreter, Procedural, Visitor
    - Observer
    - State, Strategy, Template Method

Exercises (creational patterns)

- What pattern forces a class to have a single instance?
- What patterns allow for creation of objects that are subtypes of a given type?
- What pattern helps reuse existing objects?

Exercises (creational patterns)

- Can interning be applied to mutable types?
- Can a mutable class be a Singleton?
**Factory Method**

- Maze Games are created the same way. Each MazeGame (Enchanted, Bombed) works with its own Room, Wall, and Door products.
- Factory method allows each MazeGame to create its own products (MazeGame defers creation).

```java
abstract class MazeGame {
    abstract Room createRoom();
    abstract Wall createWall();
    abstract Door createDoor();

    Maze createMaze() { ...
        Room r1 = createRoom(); Room r2 = ...
        Wall w1 = createWall(r1, r2); ... createDoor(w1); ...
    }
}
```

**Factory Method Class Diagram**

**MazeGame and Products Hierarchies**

- MazeGame
- EnchantedMazeGame
- BombedMazeGame
- Room
- Wall
- Door

**Factory Class/Object**

- Encapsulate factory methods in a factory object.
- MazeGame gives control of creation to factory object.

```java
class MazeGame {
    AbstractMazeFactory mfactory;
    MazeGame(AbstractMazeFactory mfactory) {
        this.mfactory = mfactory;
    }
    Maze createMaze() { ...
        Room r1 = mfactory.createRoom(); Room r2 = ...
        Wall w1 = mfactory.createWall(r1, r2);
        Door d1 = mfactory.createDoor(w1);
    }
}
```

**Factory Class/Object Pattern (also known as Abstract Factory)**

- Motivation: Encapsulate the factory methods into one class. Separate control over creation.

**The Prototype Pattern**

- Every object itself is a factory.
- Each class contains a `clone` method and returns a copy of the receiver object.

```java
class Room {
    Room clone() { ... }
}
```

**Using Prototypes**

```java
class MazeGame {
    Room rproto;
    Wall wproto;
    Door dproto;
    MazeGame(Room r, Wall w, Door d) {
        rproto = r; wproto = w; dproto = d;
    }
    Maze createMaze() { ...
        Room r1 = rproto.clone(); Room r2 = ...
        Wall w1 = wproto.clone();
        Door d1 = dproto.clone(); ...
    }
}
```
Singleton Pattern

Guarantees there is a single instance of the class

class Bank {
    private Bank() { … }
    private static Bank instance;
    public static Bank getInstance() {
        if (instance == null)
            instance = new Bank();
        return instance;
    }
}

Interning Pattern

Reuse existing objects with same value
To save space, to improve performance
Permitted for immutable types only
Maintain a collection of all names. If an object already exists return that object:

```java
HashMap<String, String> names;
String canonicalName(String n) {
    if (names.containsKey(n))
        return names.get(n);
    else {
        names.put(n, n);
        return n;
    }
}
```

Exercises (structural patterns)

- What design pattern represents complex whole-part objects?
- What design pattern changes the interface of a class without changing its functionality?
- What design pattern adds small pieces of functionality without changing the interface?

Wrappers

A wrapper pattern uses composition/delegation
Wrappers are a thin layer over an encapsulated class
- Modify the interface
- Extend behavior
- Restrict access
The encapsulated object (delegate) does most work
- Adapter: modifies interface, same functionality
- Decorator: same interface, extends functionality
- Proxy: same interface, same functionality

Adapter Pattern

Change an interface without changing functionality of the encapsulated class.
Reuse functionality
- Rename methods
- Convert units
- Implement a method in terms of another
**Class Adapter**

- Adapts through subclassing

```
interface Rectangle {
    void scale(int factor); // grow or shrink by factor
    float getWidth();
    float area();
}
```

**Object Adapter**

- Adapts through delegation:

```
interface Rectangle {
    void scale(int factor); // grow or shrink by factor
    float getWidth();
    float area();
}
```

**Adapter Example: Scaling Rectangles**

```
interface Rectangle {
    void scale(int factor); // grow or shrink by factor
    float getWidth();
    float area();
}

class Client {
    void clientMethod(Rectangle r) {
        ... r.scale(2);
    }
}

class NonScalableRectangle {
    void setWidth(); // no scale method!
}
```

**Class Adapter**

- Adapting via subclassing

```
class ScalableRectangle1 extends NonScalableRectangle implements Rectangle {
    void scale(int factor) {
        setWidth(factor * r.getWidth());
        setHeight(factor * r.getHeight());
    }
}
```

**Object Adapter**

- Adapting via delegation: forward to delegate

```
class ScalableRectangle2 implements Rectangle {
    NonScalableRectangle r; // delegate
    ScalableRectangle2(NonScalableRectangle r) {
        this.r = r;
    }
    void scale(int factor) {
        setWidth(factor * r.getWidth());
        setHeight(factor * r.getHeight());
    }
    float getWidth() { return r.getWidth(); }
    ...
}
```

**Structure of Decorator**

- Motivation: add small chunks of functionality without changing the interface
**Proxy Pattern**

- Same interface and functionality as the enclosed class
- Control access to other object
  - Communication: manage network details when using a remote object
  - Locking: serialize access by multiple clients
  - Security: permit access only if proper credentials
  - Creation: object might not yet exist (creation is expensive). Hide latency when creating object. Avoid work if object never used

**Graphic**

**ImageProxy**

**Editor**

```java
if (image == 0) {
  // load image
}
image.draw();
```

**Composite Pattern**

- Good for part-whole relationships
- Can represent arbitrarily complex objects

- Client treats a **composite** object (a **collection** of units) the **same** as a simple object (an **atomic** unit)

**Using Composite to represent boolean expressions**

```java
abstract class BooleanExp {
  boolean eval(Context c);
}

class Constant extends BooleanExp {
  private boolean const;
  Constant(boolean const) { this.const=const; }  
  boolean eval(Context c) { return const; }
}

class VarExp extends BooleanExp {
  String varname;
  VarExp(String var) { varname = var; }
  boolean eval(Context c) { return c.lookup(varname); }
}
```

**Using Composite to represent boolean expressions**

```java
class AndExp extends BooleanExp {
  private BooleanExp leftExp;
  private BooleanExp rightExp;
  boolean eval(Context c) { return leftExp.eval(c) && rightExp.eval(c); }
}
```

// analogous definitions for OrExp and NotExp

**Object Structure vs. Class Diagram**
Exercises (Behavioral Patterns)

- What pattern(s) help traverse composite objects?
- What pattern(s) groups unrelated traversal operations into classes in the composite hierarchy?
- What pattern(s) group all related traversal operations into separate classes?

Exercises

- If you anticipate the composite hierarchy to change and the set of operations to stay constant, what pattern would you rather use, Interpreter or Visitor?
- Conversely, if you anticipate no changes in the composite hierarchy (e.g., BooleanExp doesn’t change), but you expect addition of traversal operations, what pattern would you use, Interpreter or Visitor?

Patterns for Traversing Composites

- Interpreter pattern
  - Groups operations per class. Each class implements operations: eval, prettyPrint, etc.
  - Easy to add a class to the Composite hierarchy, hard to add a new operation

- Procedural pattern
  - Groups similar operations together

- Visitor pattern – a variation of Procedural
  - Groups operations together. Classes in composite hierarchy implement accept(Visitor)
  - Easy to add a class with operations in Visitor hierarchy, harder to add a new class in Composite hierarchy

Interpreter Pattern

Visitor Pattern

```
class VarExp extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
    class Evaluate implements Visitor {
        void visit(VarExp e) {
            // evaluates the VarExp
            v.visit(this);
        }
    }
    class AndExp extends BooleanExp {
        BooleanExp leftExp;
        BooleanExp rightExp;
        void accept(Visitor v) {
            leftExp.accept(v);
            rightExp.accept(v);
            v.visit(this);
        }
        class PrettyPrint implements Visitor {
            //...
        }
    }
    ...
}
```
The Visitor Pattern

- **Client**
  - **Constant**
    - `accept(Visitor v)`
  - **VarExp**
    - `accept(Visitor v)`
  - **NotExp**
    - `accept(Visitor v)`
  - **OrExp**
    - `accept(Visitor v)`
  - **AndExp**
    - `accept(Visitor v)`

**Visitor**
- `visit(Constant e)`
- `visit(VarExp e)`
- `visit(NotExp e)`
- `visit(AndExp e)`
- `visit(OrExp e)`

**EvaluateVisitor**
- `visit(Constant e)`
- `visit(VarExp e)`
- `visit(NotExp e)`
- `visit(AndExp e)`
- `visit(OrExp e)`

**PrettyPrintVisitor**
- `visit(Constant e)`
- `visit(VarExp e)`
- `visit(NotExp e)`
- `visit(AndExp e)`
- `visit(OrExp e)`

Exercise

- Write **Count** implements **Visitor**
  - Counts # subexpressions in a boolean expression

- Write **EvaluateVisitor** implements **Visitor**
  - Evaluates boolean expression

Façade Pattern

- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library

- Example: We want to perform secure file copies to a server. There is a powerful and complex general purpose library. What is the best way to interact with this library?

Façade Pattern

- Build a Façade to the library, to hide its (mostly irrelevant) complexity. **SecureCopy** is the Façade.

Observer Pattern

- Question: how to handle an object (model), which has many “observers” (views) that need to be notified and updated when the object changes state

- For example, an interface toolkit with various presentation formats (spreadsheet, bar chart, pie chart). When application data, e.g., stocks data (model) changes, all presentations (views) should change accordingly
A Better Design: The Observer

- Data class has minimal interaction with Views
- Only needs to update Views when it changes

Old, naive design:
```java
class Data {
    ... void updateViews() {
        spreadsheet.update(newData);
        barChart.update(newData);
        // Edit this method when
        // different views are added.
        // Bad!
    }
}
```

Better design:
```java
class Data {
    List<Observer> observers;
    void notifyObservers() {
        for (obs : observers)
            obs.update(newData);
    }
    interface Observer {
        void update(...);
    }
}
```

Push vs. Pull Model

- Question: How does the object (Data in our case) know what info each observer (View) needs?
- A push model sends all the info to Views
- A pull model does not send info directly. It gives access to the Data object to all Views and lets each View extract the data they need

Refactoring

- Premise: we have written complex (ugly) code that works. Can we simplify this code?
- Refactoring: structured, disciplined methodology for rewriting code
- Small step behavior-preserving transformations
- Followed by running test cases

Refactorings

- Extract Method, Move Method, Replace Temp with Query, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism
- Goal: achieve code that is short, tight, clear and without duplication
- Did I say this already: small change + tests

Class Diagram

```
Data

---
|
|
---
|

Observer

attach(Observer)
detach(Observer)
notify()

---
|
for (Observer obs : observers)
obs.update(...)

---
|
|
SpreadsheetView

BarChartView

---
|
|
|
|
Client is responsible for View creation:
data = new Data();
data.attach(new BarChartView());
```

Data keeps list of Views, notifies them when change. Data is minimally connected to Views!
Topics

Usability
- Definition of usability, dimensions of usability, design principles for learnability, visibility, efficiency and safety, Fitts’s law, Steering law

Usability
- Usability: how well users can use the system’s functionality
- Dimensions of usability
  - Learnability: is it easy to learn?
  - Efficiency: once learned, is it fast to use?
  - Safety: are errors few and recoverable?
  - Memorability: is it easy to remember what you learned?
  - Satisfaction: is it enjoyable to use?

Usability
- Design principles for learnability
  - Consistency: internal, external, metaphorical
  - Use simple words, not tech jargon
  - Recognition, not recall
- Design principles for visibility
  - Make system state visible
  - Give prompt feedback
- Simplicity!

Usability
- Design principles for efficiency
  - Human motor processor, Fitts’s law and Steering law:
    - Make important targets big and nearby
    - Avoid steering tasks
    - Provide shortcuts
- Design principles for safety (error handling)
  - Avoid mode errors
  - Use confirmation windows sparingly

Software Process
- Software lifecycle activities:
  - Requirements analysis
  - Design
  - Implementation
  - Integration + Testing and verification
  - Deployment and maintenance
    - Maintenance is costly. The later a problem is found, the costlier it is to fix
  - Software process puts these together
    - How do we combine these activities?
    - In what order?
Activities and Their Artifacts

- Requirements analysis produces “requirements documents”
  - Use-case model, supplementary specifications

- Design produces “design models”
  - Class diagrams, interaction diagrams, ADT specs, other

- Implementation produces, well, … obviously code
  - + specs for classes and individual methods, AFs and RIs
  - Readability of code is crucial!

- Testing produces
  - Test suites

Requirements Analysis is Hard

- Requirements analysis determines the functional and non-functional requirements of the system

- Requirements are a major causes of project failure
  - Poor user input
  - Incomplete requirements
  - Changing requirements

Classification of Requirements

- FURPS+ model

  - The FURPS:
    - Functionality, Usability, Reliability, Performance, Supportability

  - The +:
    - Design constraints, implementation requirements (e.g., must use Java), other

Requirements Analysis Artifacts

- Requirements analysis produces:
  - Use-case model
    - A set of use cases
    - Specifies the functional requirements (behavior, features) of the system

  - Supplementary specification
    - Specifies non-functional requirements (-ilities: usability, reliability, performance, supportability)

Use Cases

- Describe the interaction of the user with the system as TEXT stories

  - The most widely used approach to requirements analysis in modern software practice
    - Requirements are discovered and recorded through use cases
    - All other activities influenced by use cases!

Example Use Case

- Point-of-sale (POS) system
  - Process Sale: A customer arrives at checkout with items to buy. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt.

  - The use case is a collection of scenarios: main success scenario + scenario variations
Software Process

- Software lifecycle activities:
  - Requirements analysis
  - Design
  - Implementation
  - Testing
  - Deployment and maintenance

- Software process puts these activities together
- Software process forces attention to these activities and their artifacts

Some Software Processes

- Code-and-fix (ad-hoc): write some code, make up some inputs, debug
- Waterfall: 1st: requirements analysis, 2nd: design, 3rd: implementation, 4th: testing
- Iterative (Unified process, Agile, Scrum) repeat activities: (a small chunk of requirements, design, implementation, testing)
- Other