Reasoning About ADTs, Assertions and Exceptions
How to Design Your Code

• The hard way: Start hacking. When something doesn’t work, hack some more

• The easier (and professional) 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, less debugging
How to Verify Your Code

• The hard way: hacking, make up some inputs
• An easier way: systematic testing
  • Black-box testing techniques (more later)
  • High white-box coverage (more later)
  • Both use JUnit framework
• Also: reasoning, complementary to testing
  • Prove that code is correct
    • Implementation satisfies specification
    • Rep invariant is preserved
  • We will write informal proofs
Uses of Reasoning

• Goal: show that code is **correct**
  • Verify that the implementation satisfies its specification. Hard!
    • Forward reasoning: show that if precondition holds, postcondition holds
    • Backward reasoning: compute weakest precondition, then show stated precondition **implies** the weakest precondition
    • Reasoning is an important debugging tool

• Prove (using informal manual proofs) that rep invariant holds. This is sometimes easy, sometimes hard...
Goal: Show that Rep Invariant Is Satisfied

- Testing
  - Choose representative objects and check rep
  - Add checkRep() method that verifies representation after each method use.
  - Problem: it is often impossible to exhaustively test, therefore, we have to choose well

- Reasoning
  - Prove that all objects satisfy rep invariant
  - Sometimes easier than testing, sometimes harder
  - You should know how to use it appropriately

- Why not always leave checkRep() in code?
Ways to Make New Objects

\[ a = \text{constructor} \]

\[ b = a.\text{producer} \]

\[ a' = a.\text{mutator} \]

\[ a'' = a.\text{observer} \]

\[ c = b.\text{producer} \]

\[ b' = b.\text{mutator} \]

\[ b'' = b.\text{observer} \]

\[ \ldots \]

Very many objects but limited number of types of operations!
Verify that Rep Invariant Is Satisfied

• We can have very many objects, but limited number of operations

• How do we prove all objects satisfy rep invariant?
  • Induction!

• Consider all ways to make a new object
  • Constructors
  • Producers

• All ways to modify an existing object
  • Mutators
  • Observers, producers.
    • Should producers, observers modify the existing object?
Benevolent Side Effects in Observers

• An implementation of observer `IntSet.contains`:

```java
boolean contains(int x) {
    int i = data.indexOf(x);
    if (i == -1)
        return false;
    // move-to front optimization
    // speeds up repeated membership tests
    Integer y = data.elementAt(0);
    data.set(0, x);
    data.set(i, y);
    return true;
}
```

• Mutates rep (even though it does not change abstract value), must show rep invariant still holds!
Induction

• Proving facts about 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
  • Assumes no rep exposure

• Remember, our proofs are informal
The IntSet ADT

/** Overview: An IntSet is a **mutable** set
 * of integers. E.g., \{ x_1, x_2, \ldots x_n \}, \{\}. 
 * There are no nulls and no duplicates in the set.
 */

    // effects: makes a new empty IntSet
public IntSet()

    // modifies: this
    // effects: this_{post} = this_{pre} U \{ x \}
public void add(int x)

    // modifies: this
    // effects: this_{post} = this_{pre} - \{ x \}
public void remove(int x)

    // returns: (x in this)
public boolean contains(int x)

    // reruns: cardinality of this
public int size()
Implementation of IntSet

class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data;
    public IntSet() {
        data = new ArrayList<Integer>();
    }
    public void add(int x) {
        if (!contains(x)) data.add(x);
    }
    public void remove(int x) {
        data.remove(new Integer(x));
    }
    public boolean contains(int x) {
        return data.contains(x);
    }
}
Proof. IntSet Satisfies Rep Invariant

**Rep invariant: data has no nulls and no duplicates**

• Base case: constructor

```java
public IntSet() {
    data = new ArrayList<Integer>();
}
```

Rep invariant trivially holds

• Inductive step: for each method
  • Assume rep invariant holds on entry
  • Prove rep invariant holds on exit
Inductive Step, \textit{contains}

\textbf{Rep invariant: data has no nulls and no duplicates}

\begin{verbatim}
public boolean contains(int x) {
    return data.contains(x);
}
\end{verbatim}

• \texttt{List.contains} does not change \texttt{data}, so neither does \texttt{IntSet.contains}

• Therefore, rep invariant is preserved.

• Why do we even need to check \texttt{contains}?
contains with Benevolent Side Effects

• An implementation of observer `IntSet.contains`:

```java
boolean contains(int x) {
    int i = data.indexOf(x);
    if (i == -1)
        return false;
    // move-to front optimization
    // speeds up repeated membership tests
    Integer y = data.elementAt(0);
    data.set(0,x);
    data.set(i,y);
    return true;
}
```

• We swapped elements of `data` at positions `i` and `0`. If there were no duplicates and no nulls on entry, there are no duplicates and no nulls on exit
Inductive Step, \textit{remove}

Rep invariant: data has no nulls and no duplicates

\begin{verbatim}
public void remove(int x) {
    data.remove(new Integer(x));
}
\end{verbatim}

- \texttt{ArrayList.remove} has two behaviors
  - Removes an element
  - If there were no duplicates on entry, remove can’t change that.
  - Only addition can violate rep invariant
  - Therefore, rep invariant is preserved
Inductive Step, **add**

**Rep invariant:** data has no nulls and no duplicates

public void add(int x) {
    if (!contains(x))
        data.add(x);
}

• **Case 1:** \(x\) in \(\text{data}_{\text{pre}}\)
  • \(\text{data}\) is unchanged, thus rep invariant is preserved

• **Case 2:** \(x\) is not in \(\text{data}_{\text{pre}}\)
  • New element is not null (ints can’t be null) or a duplicate, thus rep invariant holds at exit
  • Uses autoboxing
Inductive Step, add

• How does contains determine that an object is already in array?
  • JavaDocs says:
    • Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
  • Notice that it uses equals() for contained type
  • Integer overrides equals()
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!

• Exposure of immutable rep is OK.
• Exposure of mutable rep is not!
Problem: Willy Wazoo’s IntStack

• Help Willy implement an **IntStack** with an **IntMap**

class WillysIntStack implements IntStack {
    private IntMap theRep;
    int size;
    ...

• Write a rep invariant and abstraction function
**IntMap Overview**

The Overview:

```c
/** An IntMap is a mapping from integers to integers. * It implements a subset of the functionality of Map<int,int>. * All operations are exactly as specified in the documentation for Map. *

* IntMap can be thought of as a set of key-value pairs: *

  * @specfield pairs = { <k1, v1>, <k2, v2>, <k3, v3>, ...

  */
```
IntMap Description

class IntMap {
    IntMap() {...}
    /** Associates specified value with specified key in pairs. */
    bool put(int key, int value) {...}
    /** Removes the mapping for key from pairs if it is present. */
    void remove(int key) {...}
    /** Returns true if pairs contains a mapping for the specified key. */
    bool containsKey(int key) {...}
    /** Returns the value to which specified key is mapped, or 0 if this map contains no mapping for the key. */
    int get(int key) {...}
}

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.get(size);
        theRep.remove(size);
        size = size-1;
        return val;
    }
}
Willy’s IntStack

AF: Willy’s IntStack is a LIFO collection consisting of a map between positions and data: Map <1, data1>, <2, Data2> ... <size, data_size> -> stack data1, data2...data_size

Rep invariant: size of theRep == size
Data is only accessed at size position.

Possible problem: doesn’t check for empty stack on pop.
Review Problem: 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
Practice Defensive Programming

• Check
  • Precondition
  • Postcondition
  • Rep invariant
  • Other properties we know must hold
    • Loop invariants

• Check \textit{statically} via reasoning
  • “Statically” means before execution
  • Works in simpler cases can be difficult in general
    • Motivates us to simplify and/or decompose our code!
Practice Defensive Programming

• Check **dynamically** via assertions
  • At run time
  ```java
  assert index >= 0;
  assert coeffs.length-1 == degree : “Bad rep”
  assert coeffs[degree] != 0 : “Bad rep”
  ```
  • Write assertions, as you write code
  • Not to be confused with JUnit method such as `assertEquals`!
Assertions

• java runs with assertions disabled (default)
  • Submitty run Java with assertions disabled
• java –ea runs Java with assertions enabled
• For Eclipse, see http://stackoverflow.com/questions/5509082/eclipse-enable-assertions
• Always enable assertions during development. Turn off in rare circumstances

If assertion fails, program exits:
Exception in thread "main" java.lang.AssertionError
  at Main.main(Main.java:34)

assert (index >= 0) && (index < names.length);
When NOT to Use Assertions

• Useless:

\[
x = y+1; \\
assert x == y+1;
\]

• When there are side effects

\[
assert list.remove(x);
\]

// Better:

\[
boolean found = list.remove(x);
assert found;
\]
Check Assertions

Check to see if assertions are enabled

```java
boolean assertEnabled;
int flag = 0;
assert ((flag=1) == 1);
if (flag == 1) assertEnabled = true;
else assertEnabled = false;
```

Cleaner method

```java
boolean assertsEnabled = false;
// Intentional side-effect!!!
// If assertions not enabled, nothing happens
assert assertsEnabled = true;
// Now assertsEnabled is set to the correct value
```
Failure

Some causes of failure

1. Misuse of your code
   • Precondition violation

2. Errors in your code
   • Bugs, rep exposure, many more

3. Unpredictable external problems
   • Out of memory
   • Missing file
   • Memory corruption
   • Connection failure
   • Etc.
What to Do When Something Goes Wrong?

- Fail friendly, fail early to prevent harm

- Goal 1: Give information
  - To the programmer, to the client code

- Goal 2: Prevent harm
  - Abort: inform a human, cleanup, log error, etc.
  - Retry: problem might be temporary
    - E.g. file busy
    - Not always possible, be cautious
  - Skip subcomputation: permit rest of program to continue
    - Warn user this is happening
  - Fix the problem (usually infeasible)
    - Can be dangerous
Preconditions vs. Exceptions

• A precondition tells client not to misuse your code
  • Adding preconditions *weakens the spec*

• A precondition ducks the problem
  • Behavior of your code when precondition is violated is unspecified!
  • Does not help clients violating precondition of your code

• Removing a precondition requires *specifying the new behavior*.
  • *Strengthens the spec*
    • Example: specify that an *exception* is thrown
    • Exceptions specify behavior when some constraint is violated
    • It’s almost always better to specify behavior rather than leave it unspecified
Which One Is Better?

Choice 1:
// modifies: this
// effects: removes element at index from this
// throws: IndexOutOfBoundsException if index < 0 || index >= this.size
public void remove(int index) {
    if (index >= size() || index < 0)
        throw new IndexOutOfBoundsException(“Info…”);
    else
        // remove element at index from collection
}

Choice 2:
// requires: 0 <= index < this.size
// modifies: this
// effects: removes element at index from this
public void remove(int index) {
    // no check, remove element at index
}
Preconditions vs. Exceptions

• In certain cases, a precondition is the right choice
  • When checking would be expensive. E.g., array is sorted
  • In private methods

• Whenever possible, remove preconditions from public methods and specify behavior
  • Often, this entails throwing an Exception
  • Stronger spec, easier to use by client
Square Root, With Precondition and Assertions

// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
    assert x >= 0 : "Input must be >=0";
    double result;
    ... // compute result

    return result;
}
Better: Square root, Specified for All Inputs

```java
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x) throws IllegalArgumentException {
    double result;
    if (x < 0)
        throw new IllegalArgumentException("...");
    ...
    // compute result
    return result;
}
```

CSCI 2600 Spring 2021
Better: Square root, Specified for All Inputs

Client code:

```java
try {
    y = sqrt(-1);
} catch (IllegalArgumentException e) {
    e.printStackTrace(); // or take same other action
}
```

Exception is handled by `catch` block associated with nearest dynamically enclosing `try`.

Top-level handler: print stack trace, terminate program.
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 non-local error handling
  • A method far down the call stack can handle a deep error!

<table>
<thead>
<tr>
<th>readFile</th>
<th>readLine</th>
<th>readChar</th>
<th>decodeChar</th>
<th>main</th>
</tr>
</thead>
</table>
The **finally** Block

- **finally** is always executed
  - No matter whether exception is thrown or not
- Useful for clean-up code

```java
FileWriter out = null;
try {
    out = new FileWriter(...);
    ... write to out; may throw IOException
} finally {
    if (out != null) {
        out.close();
    }
}
```
Propagating an Exception up the Call Chain

```java
// throws: IllegalArgumentException if no real solution exists
// returns: x such that ax^2 + bx + c = 0
double solveQuad(double a, double b, double c)
    throws IllegalArgumentException {
    ...
    // exception thrown by sqrt is declared,
    // no need to catch it here
    return (-b + \sqrt{b^2 - 4ac})/(2a);
}
```
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
  • Hard to propagate up call stack
  • Error-prone: programmer forgets to check result? The value is illegal and will cause problems later
  • Ugly

• Exceptions are often a better solution
Exceptions vs. Special Values

- Why exceptions?
  - Handling special values is verbose

```java
if (doSomething() == val1) {
    if (doSomethingElse() == val2) {
        if (doSomethingElseAgain() == val3) {
            // etc.
        } else {
            // react to failure of doSomethingElseAgain
        }
    } else {
        // react to failure of doSomethingElse
    }
} else {
    // react to failure of doSomething
}
```

```
try {
    doSomething() ;
    doSomethingElse() ;
    doSomethingElseAgain() ;
} catch(SomethingException e1) {
    // react to failure of doSomething
} catch(SomethingElseException e2) {
    // react to failure of doSomethingElse
} catch(SomethingElseAgainException e3) {
    // react to failure of doSomethingElseAgain
}
```
Exceptions vs. Special Values

• Return codes can cause problems when ignored.
  • Method returns null reference; reference is used later in program.

• Exceptions are typed.
  • So are special values, but a method can throw multiple types of exception
  • Methods can only return one type

• Java.lang.Math returns NaN for many standard math functions
  • NaNs are "sticky"
  • SomeType o = add(a, div(b, c)) ;
  • May be difficult to know where NaN arose

• General Rule of Thumb:
  • Throw when something should not happen
  • Return a special value when something unusual but generally expected can happen and client code can react to it.
    • Many (not all) java.lang.Math methods return NaN
    • Some post Java 7 methods throw ArithmeticException
Two Distinct Uses of Exceptions

• (External) failures (e.g., device failure)
  • Unexpected by your code
  • Usually unrecoverable. If condition is left unchecked, exception propagates up the stack

• Special results
  • Expected by your code
  • Always check and handle locally.
    • Maybe take special action and continue computing
    • May throw a module-level exception, e.g.
      • In solveQuad, catch an ArithmeticException and throw a NoRealSolutionException.
Java Exceptions: Checked vs. Unchecked Exceptions

• Checked exceptions
  • Anything that is a subclass of java.lang.Exception
    • Except for RuntimeException

• Unchecked Exceptions
  • Subclasses of java.lang.RuntimeException and Error

• Calls throwing checked exceptions need to be enclosed in a try{} block or handled in a level above in the caller of the method.
  • In that case the current method must declare that it throws the exceptions so that the callers can make appropriate arrangements to handle the exception.
Java Exceptions: Checked vs. Unchecked Exceptions

• Checked exceptions are checked at compile time.
  • The method must either handle the exception or it must specify the exception using throws keyword.

```java
// compile error – FileReader, etc. throw IOException
// IOException is a checked exception
// compiler gives unhandled exception error
class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\test\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```
Java Exceptions: Checked vs. Unchecked Exceptions

- Checked are checked at compile time.
  - The method must either handle the exception or it must specify the exception using throws keyword.
  - Compiler checks that the exception is being handled

```java
class Main {
    public static void main(String[] args) throws IOException {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\\test\\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```
Java Exceptions: Checked vs. Unchecked Exceptions

• Unchecked exceptions are not checked at compile time.
  • Exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.
  • In C++, all exceptions are unchecked
  • Checked exceptions are preferred
    • Compiler checks that exception will be handled
  • Error class exception should be used for serious problems.
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`
  • Often indicates a code problem, i.e. a bug

• From the JavaDoc documentation:

  *If a client can reasonably be expected to recover from an exception, make it a **checked** exception. If a client cannot do anything to recover from the exception, make it an **unchecked** exception*
In general, fix the problem

Should not be caught
Don’t Ignore Exceptions

• An empty catch block is poor style!
  • Often done to hide an error or get program to compile
  try {
    readFile(filename);
  } catch (IOException e) {} // do nothing on error

• At a minimum, print the exception
  } catch (IOException e) {
    e.printStackTrace();
  }
Exceptions, review

• Use an exception when
  • Checking the condition is feasible
  • Used in a broad or unpredictable context

• Use a precondition when
  • Checking would be prohibitive
    • E.g., requiring that a list is sorted
  • Used in a narrow context in which calls can be checked
Exceptions, review

• Avoid preconditions because
  • Caller may violate precondition
  • Program can fail in an uninformative or dangerous way
    • Program should fail as early as possible
  • Stronger preconditions -> Weaker specifications

• Use checked exceptions most of the time
• Handle exceptions sooner rather than later
Checked vs. Unchecked exceptions

• Unchecked exceptions are better if clients will usually write code that ensures the exception will not happen
  • The exception reflects completely unanticipated failures
• Otherwise, use a checked exception
  • Must be caught and handled – prevents program defects
  • Checked exceptions should be locally caught and handled
  • Checked exceptions that propagate long distance are bad design
    • If not caught, generates a program termination
• Java sometimes uses null or NaN as special value