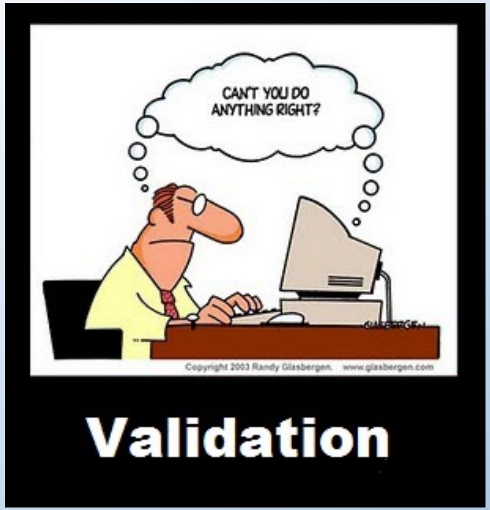


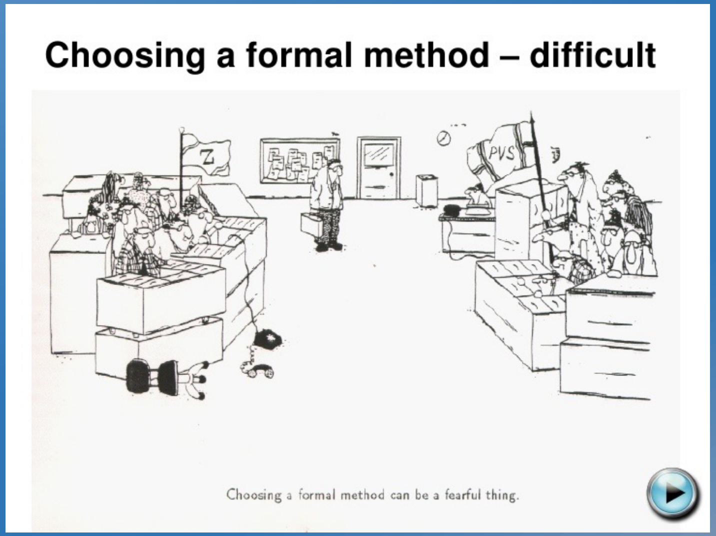




# Static Analysis I

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss



# Programming Assignment

- It should follow naming conventions
- It should follow style conventions
  - Formatting, size of methods, …
- It should be readable (block comments, etc.)
- It should include copyright, file headers

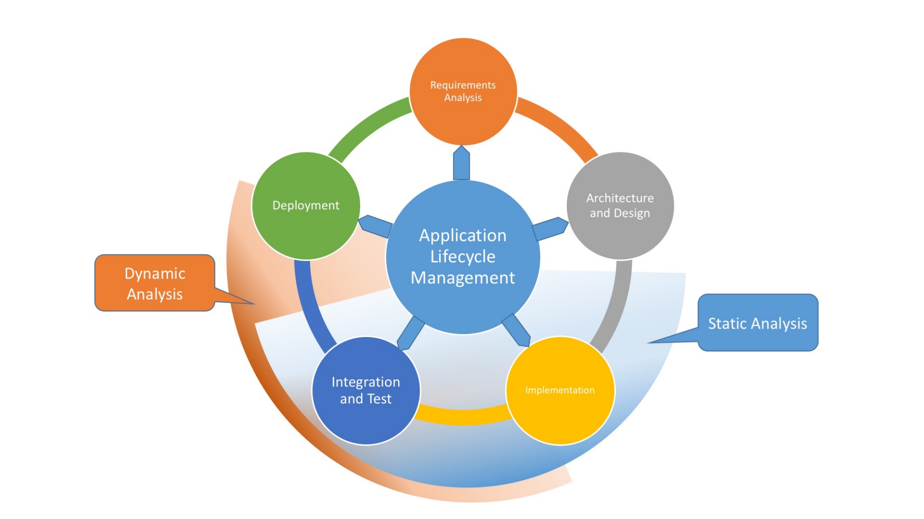
- If what you handed in doesn't do all this
  - Revise it and resubmit within the next week.

# Validation versus Verification

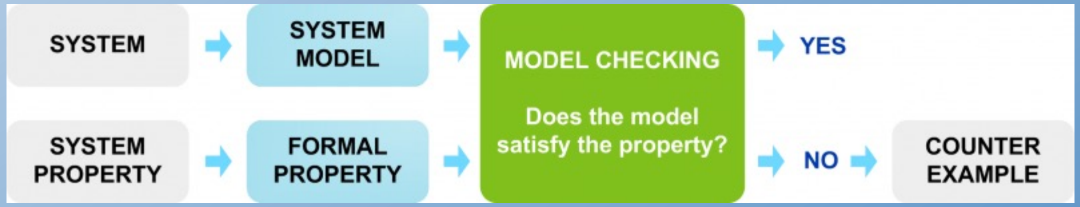
- Validation checks the program
  - Testing, code reviews, user experience, …
  - Confirmation by examination of objective evidence
    - That the program meets its requirements
- Verification
  - Proves the program meets its requirements
  - Showing it works for all possible inputs
  - Showing it works under all possible conditions
  - This is what static analysis tries to do
- Verification is needed in safety-critical systems
  - Verification is useful in security-critical systems
  - Verification is useful in general

# Goals of Static Analysis

- Demonstrate that the system will work under all conditions
  - Can work on a model of the system rather than actual code
    - As long as working on model implies working on the system
    - Note: not working in the model does not imply not working in the system
  - Need to define what ”work” means in a formal sense
- Find potential bugs without running the software
  - Considering all possible inputs rather than just samples
  - Considering all possible executions
- Checking for potential problems
  - Looking at specific problems
    - Either general or program-specific
  - Seeing if they are present in the code



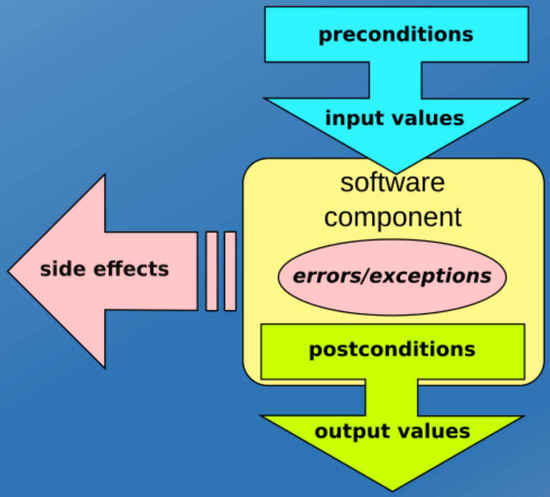
# Partial Correctness



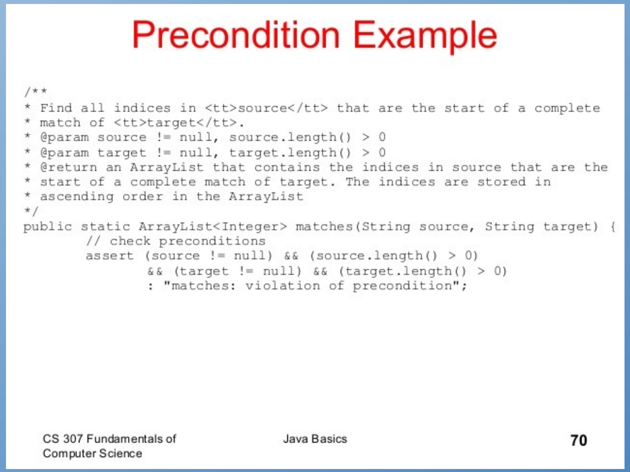

- Complete program correctness is difficult
  - Would need a complete specification of the program
  - For a large, complex system, especially an interactive one
    - Writing the specification is more difficult than writing the code
    - The specification is as or more error prone than the code
- Solution is partial correctness
  - Identify key properties the code should have
    - For safety, for security, for operation
  - Prove these properties hold in the program
    - Under **all** possible inputs and **all** possible executions
- Differs from partial correction in program theorem proving

# Contracts: The Traditional Approach

- Contracts are local formal specifications of program behavior
  - Several forms and levels
  - Partial specifications (not necessarily complete)
  - Developed by Bertrand Meyer in the language Eiffel
    - Based on formalisms developed by Dijkstra
  - Can be safety conditions or just program conditions
- Three types: Preconditions, Postconditions, Class Contracts
- Precondition contracts
  - Specify constraints on the inputs to a method or function
    - Argument can't be null; 0 < value <= 10; …
  - These are the assumptions a method makes
    - What you might check using defensive coding & documentation
  - Preconditions make these assumptions explicit



# Specifying Preconditions in Java



## Precondition Example

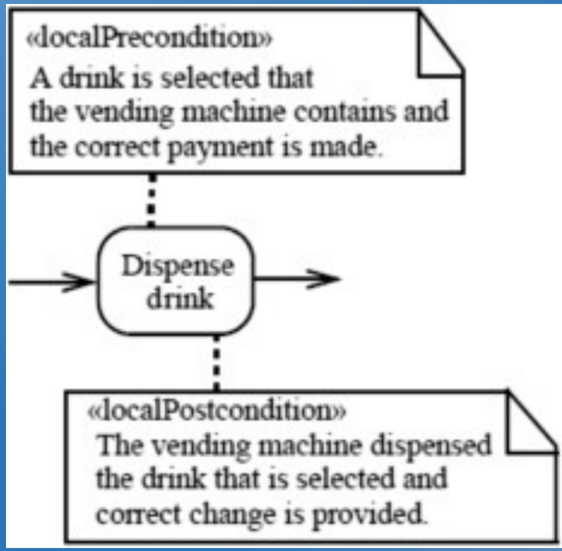
```java
/**
* Find all indices in <tt>source</tt> that are the start of a complete
* match of <tt>target</tt>.
* @param source != null, source.length() > 0
* @param target != null, target.length() > 0
* @return an ArrayList that contains the indices in source that are the
* start of a complete match of target. The indices are stored in
* ascending order in the ArrayList
*/
public static ArrayList<Integer> matches(String source, String target) {
        // check preconditions
        assert (source != null) && (source.length() > 0)
                && (target != null) && (target.length() > 0)
                : "matches: violation of precondition";
```

CS 307 Fundamentals of Computer Science Java Basics 70

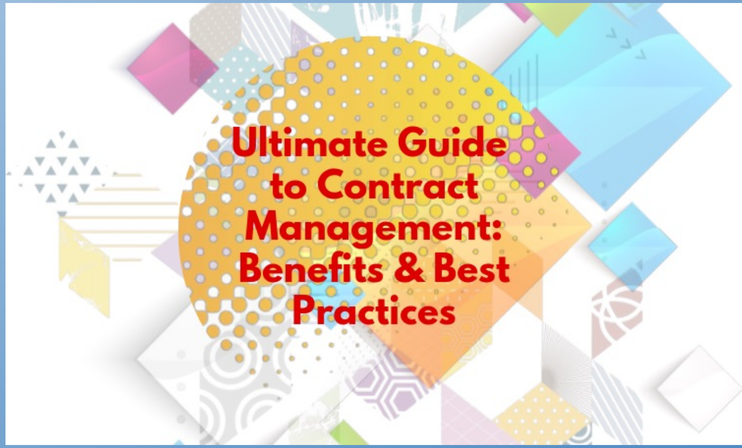
- Can be done using assert statements
  - Placed at the start of a method
  - These generate an exception if condition is violated
  - But can be ignored (compiler or execution option)
- Can be done using annotations
  - com.google.java.contract.* is one example
  - @Requires(<boolean expression>) annotation on a method
    - Expression has access to the various arguments
    - And any other accessible values
  - A bit messy since Java annotations apply to types, not methods
  - Needs to be checked outside the language
- Similar facilities exist for other languages

# Postcondition Contracts

- Specify constraints on the outputs of a method or function
  - In terms of the inputs and the return value or exception
    - Can use its own local variables if needed
  - result >= 0;  set.contains(input)
- These can be a definition of what the method does
  - But generally, are only constraints, not full specifications
- Can be specified using assert
  - With assert (…) inserted before any return statement
- Can be specified with annotations
  - @Ensures(<condition>)
  - @ThrowEnsures(<condition>)



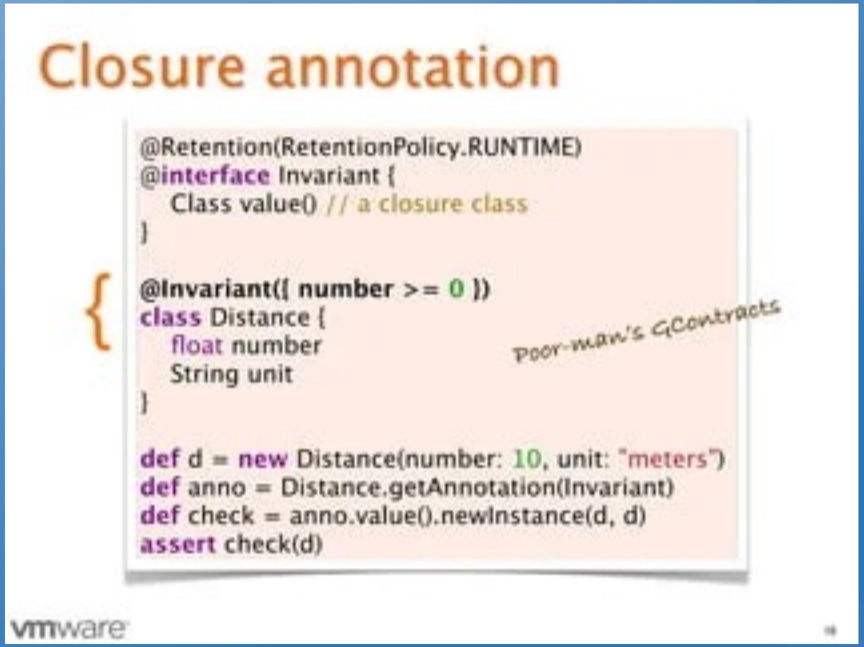
# Class Contracts




- Specify properties of a class
  - Define relationships among fields of the class
    - Constraints on the class
  - Hold whenever a public or package-protected method returns
    - Also, when a constructor returns
    - This allows inconsistent states while computing, but that are not exposed
    - Requires accurate specification of protection levels
  - Can specify usage rules for methods
    - But this is non-trivial (need to create automata as part of the check)
    - Can introduce contract variables for this purpose
- Can be defined using assert statements
  - assert balanced(); assert field != null
  - Add this at the end of each appropriate method and constructor
    - Call a method to do the check rather than check in place

# Class Contracts

- Easier to define using annotations
  - @Invariant(<condition>) specified once for a class
  - But this needs to be checked



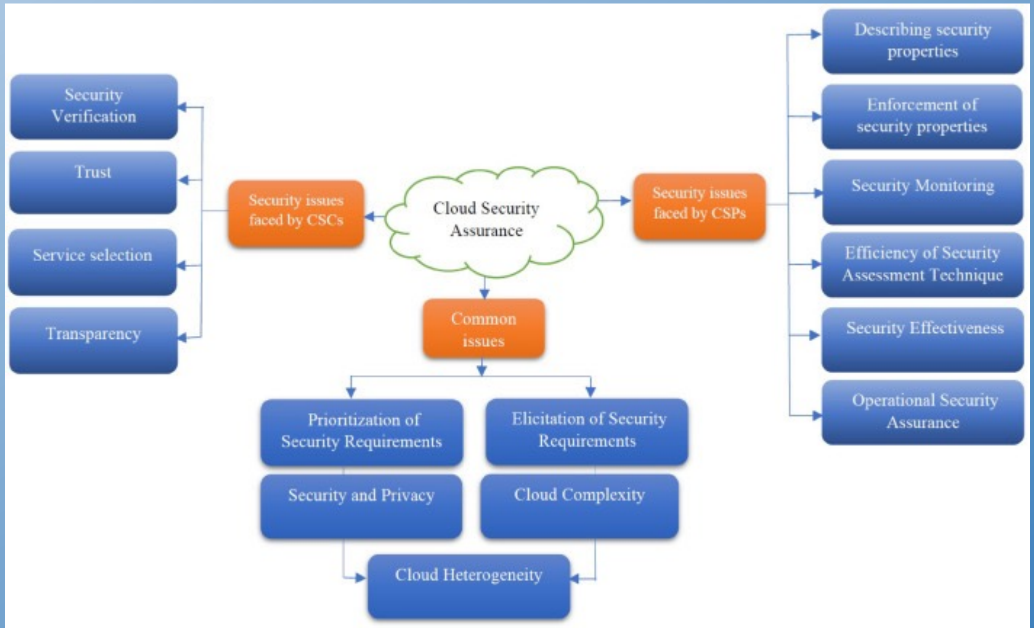
# Checking Class Contracts


- Annotations can be used to generate internal code
  - Checked at run time using annotation processing (cofoja: defunct)
  - Checked at run time using aspect-oriented programming (Oval; defunct)
  - Check at run time using class loader patching (Jass; defunct)
- Can be checked statically with proper framework
  - Proving the contract holds under all executions
    - Post condition holds given precondition
    - Class condition holds whenever a public method returns (assuming pre & post conditions)
    - Using theorem proving technology
  - Java Modeling Language (JML) did this (defunct)
  - C-spec for C# (Microsoft) does this
- Ideally contracts would be compiled in as run time checks by the language
  - But annotation processing in Java doesn't allow this
  - Compilers are starting to support some of more useful annotations

# Discussion

- Contracts seem to be a way of ensuring program behavior
  - Relatively easy to specify, non-obtrusive
- Yet they are not widely used.
- WHY?
  - Do any of you use them?
  - Would you use them if they were available?
  - Would you use them for documentation?
  - Would you use them if they were checked
  - Would you use them if they didn't cause performance problems?
  - Would you prefer them to defensive coding?
  - How difficult to you think they are to create?
  - How complete do you think they generally are?
  - Can they express global properties, safety, security?

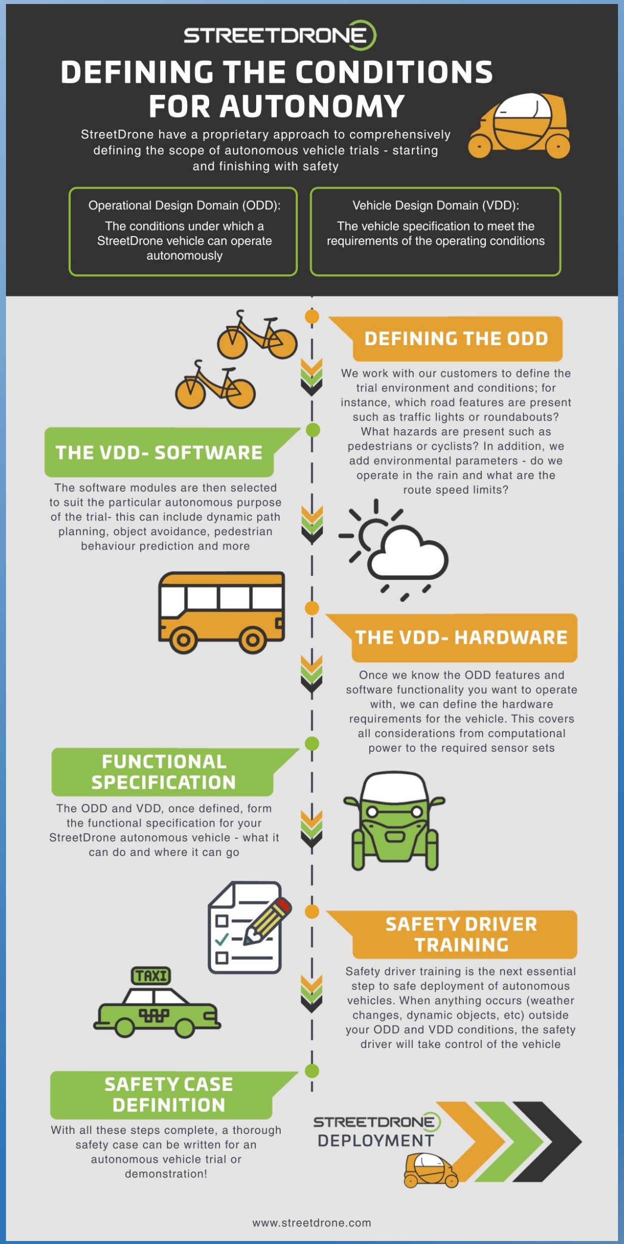
# Safety Condition




- We want to prove the whole system is secure
  - With respect to a given condition or property
- Actually, we try to prove the system is not secure
  - Define a condition that should not occur
    - A violation of safety or security
    - The program can cause two trains to crash
    - User input can flow to a SQL query
    - A user can gain super-user access without authentication
  - Prove the program can cause this condition to occur
- The proof then lets you see why the program fails
  - Proof effectively creates an example that fails
    - Shows how and why the program is unsafe
    - Tells you what to fix (can be the program or how the proof was done)
  - Failing to prove the system is not safe means the system is safe
- A condition that we use to show the system is not safe
  - Is called a **SAFETY CONDTION**

# Safety Conditions

- These are used to define the possible problems
- Defining things the program shouldn't do
  - SHORE: allowing two trains to crash
  - Writing sensitive data out
  - Invoking a SQL query with tainted data
  - Executing privileged command without privilege
  - Overflowing a buffer
- Ensuring that the system doesn't do these
  - Under any conditions
  - Under any possible sequence of inputs
  - Under any possible sequence of actions
    - Including thread schedules



# Security Problems as Safety Conditions

- Checking that tainted data doesn't go to the wrong place
  - SQL injection attacks
  - XSS attacks
  - Credit card numbers not saved
  - Secure data remains at the proper security level
- Checking there are no buffer overflow errors
- Checking that assumptions are met
  - User authorized as admin to execute admin functionality

# System Problems as Safety Conditions

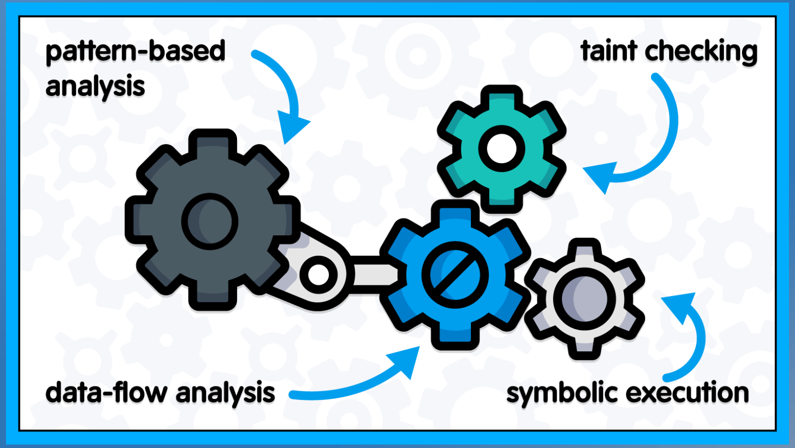
- Many correctness properties of a system
  - Can be expressed as safety conditions
  - Example: transfer of money from one account to another
  - Example: hasNext called before next
- Can express the critical system issues
- Can show the system does what it should
- But you need to state the conditions
  - These are application specific

# Safety Conditions in Embedded Systems


- Used extensively in embedded systems
  - Often these are critical systems
    - With real-world consequences
- Examples:
  - SHORE: two trains shouldn't be in the same block
  - PINBALL: the program shouldn't blow a fuse
  - A self-driving car should not crash into a person
- NASA is a big proponent

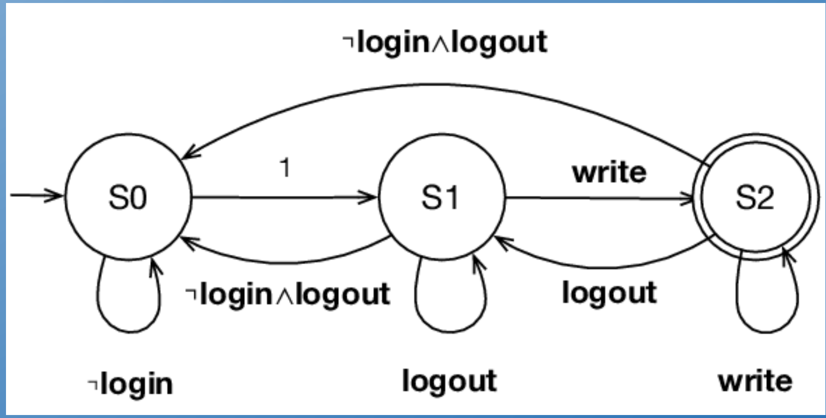
# Static Analysis Checks Safety Conditions

- Shows how a safety condition can fail (the program is bad)
  - Or shows it can't and thus can't be triggered by the program
  - When it does fail, it shows how the failure can occur
- To do static analysis effectively we need to simplify
  - The condition needs to be stated in formal finite terms
  - The program needs to be stated in finite terms
    - And tractable (problems solvable)
    - This means finite state and small finite variables bounds
- Then we can do a proof
  - Either in terms of finite automata
  - Or in terms of Boolean formula



# Büchi Automata

- Safety conditions can be specified using automata
  - Over infinite inputs
  - Transitions indicated by program events
    - Calling or returning from a method
    - Variable having a certain value at a given point
    - Execution reaching a given point
    - Creation of an object
  - Transitions indicated by predicates
    - Based on program variables
  - Safety condition expressed in terms of different states
  - Start state indicates initial state of the system
  - Error states define states that should not be reached




- Goal: Find a valid series of program events or variable settings
  - That puts the system into an error state when started in the start state
  - Again, trying to get the system to fail (not show success)

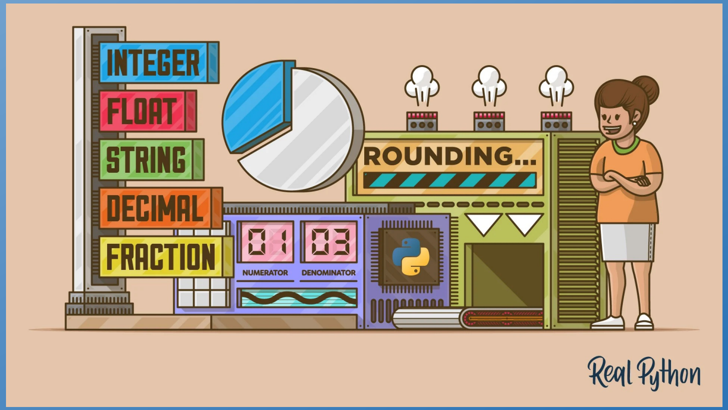
# Temporal Logical Formulas

- Can also be represented using temporal logic
- Boolean logic
  - With additional operators for time
    - X eventually occurs
    - X always occurs
    - X occurs after Y
  - Using properties of the program as a base
    - Program events, variable values, …
- Roughly equivalent to automata
  - Can map back and forth
- Other representations are also used for special cases
  - Contracts for local behaviors
  - Special structures for escape analysis

| Operator | Meaning | Example |
|---|---|---|
| $G$ | globally / forever / always | $G\,p$<br>formula $p$ is true forever<br>(that is in all states) |
| $X$ | next | $X\,p$<br>formula $p$ is true in the next state |
| $F$ | finally / sometimes | $F\,p$<br>formula $p$ is true finally<br>(that is in some states) |

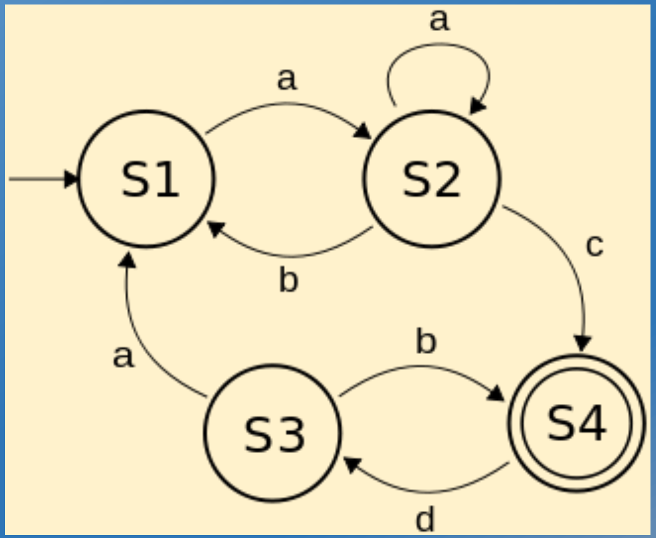
# Representing the Program’s Data

- To check these properties, the program needs to be abstracted out of the code
  - Program must be finite state (to allow decidability)
  - Essentially an extended finite automata (conditions on transitions)
  - Minor extensions (e.g., call-return) are possible, but not simple
  - Fixed number of threads if threads are considered
- The program’s data must also be finite
  - Integers, doubles, etc. need to be restricted to finite sets
    - A few significant values or ranges; then combine all the others (0,1,2, >2)
    - Need to do arithmetic with the combined values
    - Operations might yield multiple results
  - Structures
    - Represent as finite graphs (of links)
    - Again, with nodes representing an arbitrary graph
    - Fixed maximum size for arrays, lists, etc.
- Note these are approximations (generally include all possible cases)
  - Up to a point (max size, max threads)
- A lot of this can be automated



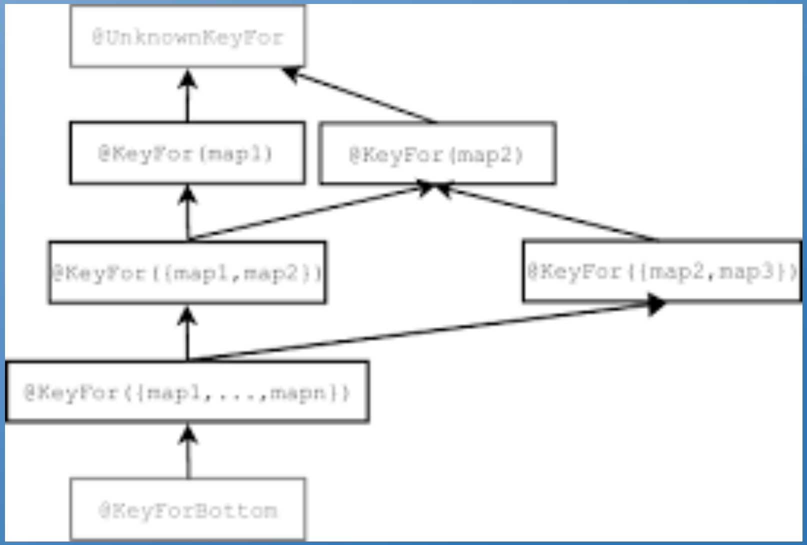
# Representing the Program

- Need to map code to an augmented finite state automata
  - States represent program locations
  - Can have a set of (finite) values for variables at each location
  - Arcs can be conditional on the values
- Generally, just use the control flow graph directly
  - Without considering most conditionals
  - Program can go either way at a conditional
- Can consider threads and thread interleavings
  - State is cross-product of states of individual threads
  - This makes the problem more difficult
- Try to make this a conservative approximation
  - It includes all possible cases
  - If problem doesn't occur in model, it can't occur in program
- Most of these mappings can be automated



# Data-Based Safety Conditions

- Some safety conditions relate to specific data items
  - Need to be checked for each logical instance of item
  - Separate automata defined for each item
    - States relate to states of the data item
    - These are equivalent to **subtypes**
  - Subtypes are type qualifiers
    - Specify the automata state for the type
    - Can be arranged in a lattice
  - Changes relate to actions on the data item
    - Creation, operations, …




- Examples
  - Iterators: Initialized, hasNext  available, hasNext none, next Called, error
  - SQL Data: SQLuntainted, SQLtainted, SQLfullytainted, Unknown

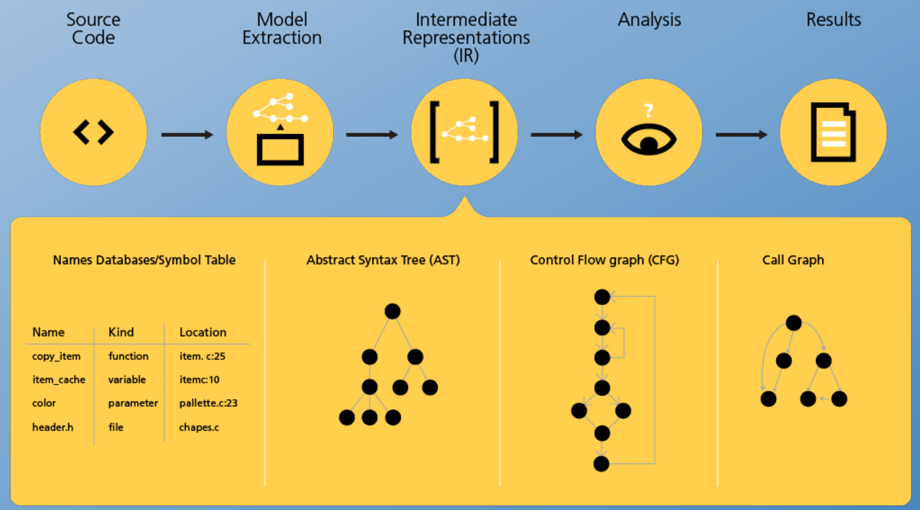
# Program-State-Based Safety Conditions

- Other conditions depend on the program overall
- Has the user been authenticated
  - Anywhere that authentication is needed
- Is a user authorized for this action
  - Authorized at the right level
- Security Level
  - Is the users access level >= that of the data

# Testing Safety Conditions

- You want to ensure safety conditions are never violated
  - Actually you want to find a violation of the safety condition
  - In the program now or in the future
  - Under any possible conditions
  - Under all possible executions
- You could add code to check this
  - But that code might be buggy or incomplete
    - Still might be a good idea: defensive coding
  - So even with code, you can't be sure there are not holes
- Static Analysis tries to be the answer
  - Can check a specific property
  - Can check it practically for all possible executions and inputs

# Static Analysis




- Defining the condition(s) to check
  - Different approaches are used
    - Subtyping (Finite state data models)
    - Finite state models (Büchi automata; temporal logic)
    - Bounds checking (variable ranges for buffer overflow)
    - Condition-specialized structures (e.g., for escape analysis)
  - Chosen mainly for convenience
    - Based on the safety condition to check
    - Based on means for checking that problem
- Static analysis then checks these for all possible
  - Inputs and valid program executions
  - Possibly assuming a simplified version of the program
    - That is a generalization
- One approach that is used here is subtype checking

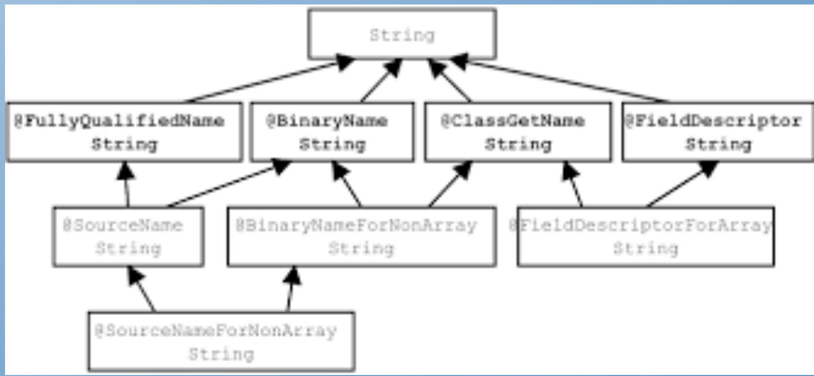
# Subtype Checking

| | |
|---|---|
| P ::=($\overline{\mathbf{L}}$, t, FM) | *CFJ program (SPL)* |
| L ::=class C extends C { $\overline{\mathbf{C\ f}}$; K $\overline{\mathbf{M}}$ } | *class decl.* |
| K ::=C($\overline{\mathbf{C\ f}}$) { super($\overline{\mathbf{f}}$); $\overline{\mathbf{this.f=f}}$; } | *constructor decl.* |
| M::=C m($\overline{\mathbf{C\ x}}$) { return t; } | *method decl.* |
| t ::= | *terms:* |
| x | *variable* |
| t.f | *field access* |
| t.m($\bar{\mathbf{t}}$) | *method invocation* |
| new C($\bar{\mathbf{t}}$) | *object creation* |
| (C)t | *cast* |

Fig. 2. CFJ Syntax

- Type checking can catch potential errors
  - This is why I encourage using strong typing
  - Finer type checking can catch more
    - Knowing X is in inches rather than centimeters
    - Knowing X may or may not be null
    - Knowing X can be tainted
- This can be done by adding subtypes to existing Java types
  - Subtype of double-inches
  - Subtype of String-nonnull or String-nullable
  - Subtype of String-tainted

# Defining Subtypes




- Subtypes in Java can be defined using type annotations
  - @NonNull, @Tainted, @Unit("inches")
  - Can be applied to the types of fields, arguments, local variables, returns
    - Any place a type is specified, can annotate that type with appropriate subtypes
- Need rules for checking and propagating subtypes
  - Assigning null to a @NonNull variable is an error
  - Passing inches to a method wanting centimeters is an error
  - Concatenating tainted data with untainted data yields tainted data
  - Inches * inches = inches$^2$; inches*scaler = inches
  - Writing tainted data to a file is an error
- These must be defined for each subtype and operator
  - But in general, the rules follow a hierarchy for the subtype
    - Thus, the specification can be simplified and practical

# Subtypes for Security Checking

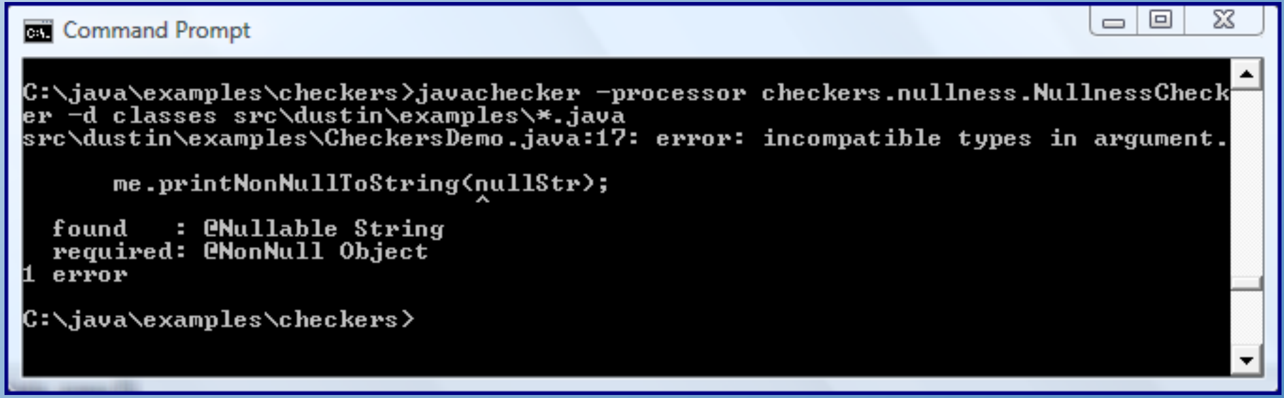
- @SqlTainted
  - Whether data is tainted with respect to SQL injection
- @HtmlTainted
  - Whether data is tainted with respect to XSS
- @FileTainted
  - Is the file name tainted to give outside access
- @Initialized
  - Has this data been initialized

# Subtype Checking Tools

Type Tools

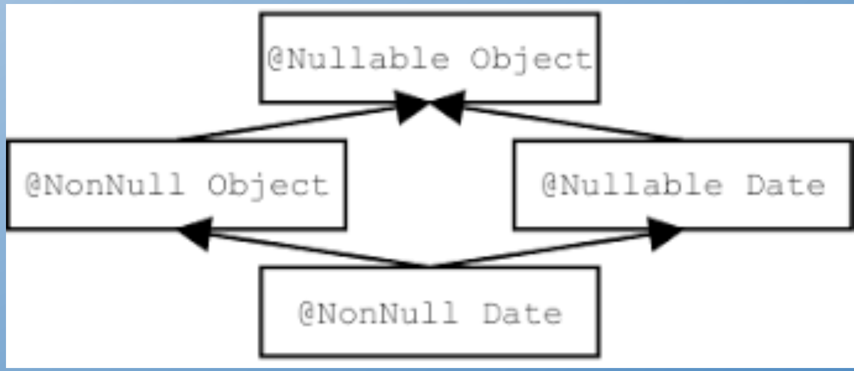
- Tools exist for checking these subtypes
  - Using type-checking technology
- Checker system
  - Available as a plug-in for IDEs or standalone
  - Runs in batch mode
- Checking is done statically
  - Check all assignments (explicit and implicit)
  - Types are propagated inside a method
    - Checking defines the target subtype at each program point
  - Types are not propagated between methods, but are checked at each boundary
    - Method call, return value
- Errors
  - No errors detected says program is correct with respect to annotations
  - Errors detected says program **might be** wrong
    - And type checking yields the execution sequence yielding the error
  - Again, this is an approximation since type combinations might not be possible

# Using Checker



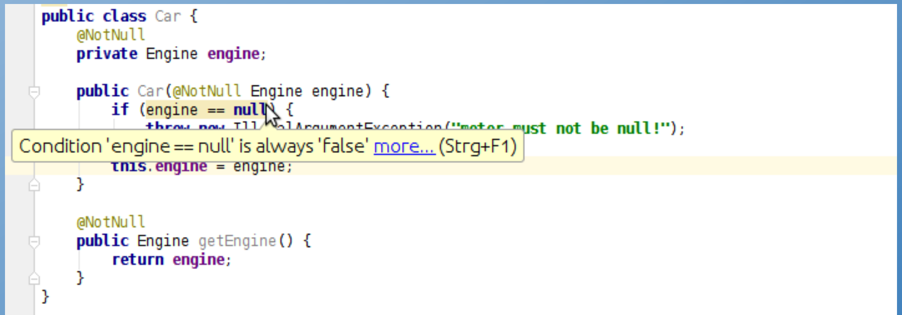

- More difficult than it looks
  - All intermediate values need to be subtyped correctly
    - Types must be defined at each boundary (method definition and call)
    - Can require a lot of annotations
    - Will miss some things (e.g., common routines)
  - Requires understanding and annotating collections
    - Saying that no value in a collection is null
    - Saying that access to a map will return something non-null
      - That this variable holds a valid key – there's an annotation for that
      - But this requires adding a lot more annotations
  - Requires annotating third-party libraries
    - Java standard library is annotated by Checker
    - But systems often use other libraries as well
  - Only checks source code, not binaries
- Nice framework, not that widely used

# Null Checking




- Checking for nulls is an important instance
  - Prevents potential NullPointerExceptions
  - Can be quite useful
- Today's compilers try to do this
  - But there are usually too many false positives
    - In constructors and routines called by constructors
    - When using collections
    - Based on library routines
  - Also, tools like FindBugs try to do this
    - Again, with lots of potential false positives
  - Can also be done using Checker with lots of annotations

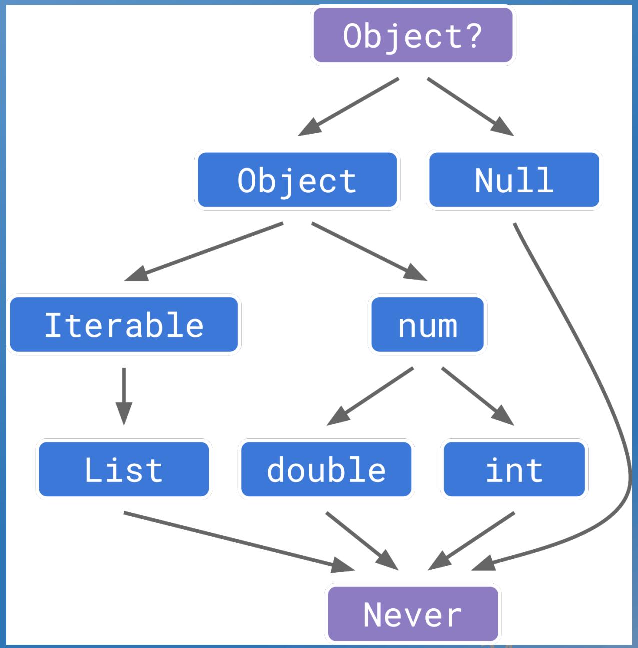
# Null Checking in the Compiler & FindBugs




- Rather than checking everywhere
  - Only check where it can implicitly be null
    - After the user has checked for null
    - After the value has been set to null and not reset on all paths
  - This can have a lot of false negatives
- This can be augmented with @NonNull, @Nullable
  - To get more accurate results
  - But these aren't necessarily supported by the compiler
- Or by building null checking into the language
  - DART: type system includes nullable/non-null explicitly

# Null Safety in Dart

- Dart includes nullability in types system
  - Either absolute
  - Or settable in the constructor
- And operators that handle null values automatically
  - With well-defined semantics
- This solves many problems
  - Catches potential errors
  - Forces user to think about null values
- But not all problems
  - Delayed initializations can still cause problems
  - Programmers get lazy and declare values as nullable
  - Compiler check based on conditionals not perfect
  - You should still except come errors



# Next Time


- Other approaches to safety checking
- Abstract interpretation
  - Using data flow analysis to simulate a high-level execution
  - Looking at all possible inputs and executions
- Model Checking
  - Mapping the safety condition and the program to finite automata
  - Developing a formula stating the safety condition fails
  - Proving that this formula is true to find counterexample
    - Or prove there is no counterexample

# Homework

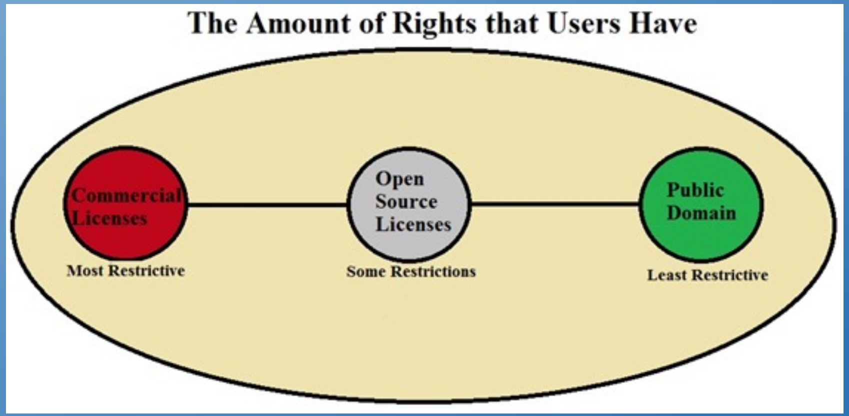
- Think about what can go wrong with your project
  - That you want to ensure the code doesn't do
  - What are the potential problems
    - Safety, security, or operational
  - What are the critical aspects of your software
  - How would you show your software works
- List one safety condition for you project
  - Submit to canvas
  - Due Thursday 11/14

# PROJECT

- Did any project want to present on 11/26?
- We will have 3½ classes devoted to project presentations
  - (11/26), 12/3, 12/5, 12/10, (12/12)
  - Each team gets <=40 minutes (half a class)
  - Last class: 12/12 can be used for final demo (<=10 minutes)
- Presentation
  - Requirements, specifications, design, implementation
    - Live demo or video (if possible; canned demo otherwise)
    - This can be saved for the last class, but would be nice to see current state
  - Possible plans for maintenance and evolution
    - Where is the project going after the class is over
  - Experience: What you learned, what you should have done better
  - Questions from the audience

# Copyright

- You should have a copyright notice at the top of each file
  - With an appropriate license
  - Or "All Rights Reserved"
  - Even for programming assignments
- In addition to your name and year
- Should be the same throughout the project
- Be aware of copyright on anything you use
  - Include in source if you borrowed code
  - And copyleft (GPL)
- Should be below block comment for the file



# Subtypes for a Type Form a Lattice