

The slide features a light gray gradient background. In the top-left and bottom-right corners, there are clusters of realistic, 3D-rendered water droplets of various sizes, some overlapping. The text is centered in a bold, black, sans-serif font.

# **CS1632, LECTURE 2: TESTING THEORY AND TERMINOLOGY**

# KEY (🔑) CONCEPT TO THE COURSE

EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR

# EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR

You need to know what “should” happen under some circumstances, then check to see if that behavior actually occurred.

For example, assume I have a function `foo`, which accepts an integer, `a`, and returns a float. What should happen if I send in the value `a = 42`?

This is a simple idea, but it’s the “Fundamental Theorem of Testing” (although note that we may violate it later...)

## EXAMPLE

Assume `foo` is supposed to return the square root of the passed in value `a`.

When I send in the value `a = 42`, then I expect to be returned the value `6.48074069841`.

When I send in the value `a = 9`, then I expect to be returned the value `3`.

When I send in the value `a = -1`, then I expect....

# THE IMPOSSIBILITY OF EXHAUSTIVE TESTING

- Let's say we want to ensure that our square root method will never fail, no matter what we send in. Assume we are using a standard Java int (signed 32-bit integer)
- How many values do we have to test?

4,294,967,296

# WHAT ABOUT A MEDIUM-SIZED, 1 000-METHOD JAVA PROGRAM?

- Assume that each method accepts one 32-bit int argument and returns one primitive value.
- If we have references to objects, or multiple arguments, etc., then the program have even more possibilities to test.
- Remember that methods in a java-like language could theoretically influence other methods (e.g., setting global variables, calling other methods, mutating objects, etc.)

4,294,967,296 ^ 1000



# THAT'S EQUAL TO...

911719507852790025097233389675787454799158467041610932663168585865905618939718996696185857419699285233872077457614691328001979817518863144393675517813756223551824941327841222420659045  
719231252962679771566322311625132454307035597053022521561246898546680858024365954586897143119190611638212638342763480050687188258215390720171988441441886648791215759762725389366216557  
446824322870031468591582748390582107692331553078297956170965592272693230942153795374906434644294709320337977283581908983329406571317899264697722812061019380899236287721753962035495230  
692778928668862404748824570782188307505182107784007030038445100444660699812696177811947887576560255655908396845270435452389665755734035596879513269638928388180406608507119416909654643  
032967159977353151143932983146397659063257169169022650584852691127482320552832649161374260954014992260158010974842315012361243480806037745824830947865010104114916409907198088842842136  
854617717944749552713213785771108961288811831237666379486767813546188006643524048639445295652978852963383737850760241163514382716651875374555699140067691456071748271087590362173711860  
044704084625336091543500691714323919052366981476276437065678838563998878381885773000341053862602758014572499348041757394660676973509282166930632955880631414046371713765667379841266106  
690291293383395081456948512017486798190459730596356276188049711558419441692878068723403972993776051906204692990827029933269512641490378794107236777670850856194442404382536670006178148  
067923537408135936520818300443759069700073839716901750673811649685604898639120754630991761876244494755004573365431289165536317975875329363610604687315411060583328683862275345757575463  
240556324712123542050306870593261281242373060461770747490826720243989627915091819824413031215358212115365367031228798361900436098033593939317108507725449460576386728537579974232974920  
375624778180923231343430936921083911139534659465800121513010419901123405611013130169487104000337304071364153665375401392391571941559516894804009997616840779274669921901896730145691283  
224737330538959493641343438083324990599872647828146559491250565907817965564228047463516269798592276478265498502288667448764514477511517488432996198979728103818498694322921561024966850  
314625907082114914714792967131863609399305526164625466885215225622397418654703382066432463542697829858218828558971223097560032462541759341803857231791197758371764898876357513845527975  
297983959090297946513522186617719293774695529057040663935309335892299198791524237701995080418148253222726701943143791040308457583407799395569470968122500963439811675572475959611793311  
581144998431728888764229631959149698707010011920915003485622492705253249322497598359968549324675486690221367445483306326155451753510122182825002506100874796508645544973831874770979358  
620234214731717114846419684079934970832838316901532604916777028567298028388862349683564295970509073047685453156724695150220004343366507024829844618597115522445867369543628493549104307  
840860011375745240897766543930223155885934303161269392094965398324965354058673841775402252237117553607196719092150177240692620980023992930495414830641411644455664330843363657287811979  
996492743740691150412105301771458223402797056844529954715900646233754267861944256911059891234426533672099268369543158016134038936671758535029408020838733922701531152110598062266053398  
102540988114340055067056602385839725309342036264566674470502804264344071460844638445576762313761453684650760417338189366245092025790481281184061913070955640388506005792845049176989863  
842887692074866493269762020830590249133982593101379509251034117211837245861653473607412360243912270294224195508080573853152865108024214615389581662126376990386719748417811864171855832  
925998262731571244736642792041456143356739457308114036780697679783341964181398779863871025415770527055336426195241935221037058819757293995048374550819894222334319683340221385890807072  
484235545990481780316101856266200860564034686501297109898065953529438125588949324225847226348909310812433561617361899152339681299074390490156912222602274432246111115008070981518368021  
943712331574489358598419626962386866787045661979781521424938788873043683915426522906643101327655071455054630762957165012891096266194047811993915980675768602419877302809260185529189805  
062249766135620116368437885017203739942121506824556134332528535659201259131178455039313741535904291557845315362022349160552766464387511801543283280605319811115123251522965371345440776

The image features a light gray gradient background. In the top-left and bottom-right corners, there are several realistic-looking water droplets of various sizes, some overlapping. The text is centered in the upper half of the image.


**WOULD HAVING THAT MANY TESTS GUARANTEE  
THAT THERE ARE NO PROBLEMS WITH THE  
SYSTEM UNDER TEST?**

# LOL NOPE

- Data races?
- Compiler issues?
- Non-functional issues (performance, usability, etc.)?
- Floating-point issues?
- Integration issues?
- Systems-level issues?
- **Ambiguous or misunderstood requirements?**



# TESTING = ART + SCIENCE

- There are techniques for testing which can reduce the number of tests necessary for sufficient test coverage.
  - We will need to define what we mean by “sufficient test coverage”.
  - We will also require domain knowledge.
- 

# EQUIVALENCE CLASS PARTITIONING

- We can partition the testing parameters into “equivalence classes”
  - Equivalence class = a natural grouping of values with similar behavior
- For example, in our square root method:
  - Negative numbers  $\rightarrow$  Imaginary numbers (or exception)
  - 0  $\rightarrow$  0
  - Positive numbers  $\rightarrow$  Positive numbers

# EQUIVALENCE CLASSES ARE STRICTLY PARTITIONED

- For any given input value, it must belong to one and **ONLY** one equivalence class (strictly partitioned)
  - If there are values that seem like they belong in multiple equivalence classes, you either need:
    - Multiple partitionings
    - Another equivalence class

# EXAMPLE

- Assume you have a program which will return the square root of an int, and if the number is whole (e.g., 1 or 2, but not 1.342), it should print it out in **red**, otherwise it will print it out in black.
- You can have two partitionings:
  - (the positive/0/negative partitioning on the previous slide)
  - Another partitioning:
    - Number is whole -> output printed in **red**
    - Number is not whole -> output printed in black
- Therefore, for every value, there are multiple partitionings to check

# THEY DO NOT HAVE TO BE NUMERIC

- On Twitter, if you follow somebody, you see all of their tweets, unless they are writing directly to somebody you do not follow.
- Equivalence classes:
  - You do not follow person A -> DO NOT see the tweet
  - You do follow person A, they are not writing directly to somebody -> see the tweet
  - You do follow person A, they are writing directly to person B, whom you also follow -> see the tweet
  - You do follow person A, they are writing directly to person B, whom do you not follow -> DO NOT see tweet



# TEST EACH EQUIVALENCE CLASS

- Pick at least one value from each equivalence class
- This will ensure you capture behavior from each “class” of possible behavior
- Will find a good percentage of defects without exhaustive testing!
- We reduced the problem something a human can do! Woo-hoo!
- How to pick the input? Well, that is part of the art.
  - However, there are some good guidelines!

## INTERIOR AND BOUNDARY VALUES

- Theory: Problems are more prevalent on the boundaries of equivalence classes than in the middle.

# WHY?

```
public boolean canBePresidentOfUnitedStates(  
    boolean naturalBornCitizen,  
    int age) {  
    return naturalBornCitizen && age > 35;  
}
```

# EQUIVALENCE CLASS PARTITIONING

CANNOT\_BE\_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN\_BE\_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64...  
.INFINITY]

# WHERE ARE PROBLEMS LIKELY?

CANNOT\_BE\_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN\_BE\_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64...  
.INFINITY]

# TRY TO ENSURE THAT YOU TEST BOUNDARY AND INTERIOR VALUES

CANNOT\_BE\_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN\_BE\_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64...  
.INFINITY]

- Are we missing anything?

# “HIDDEN” (IMPLICIT) BOUNDARY VALUES

- The boundary values we have gone over already are explicit – that is, they are defined, or at least able to be deduced from, the requirements of the problem itself.
- Some boundaries are implicit – they are generated from the domain, architecture, hardware, or other elements:
  - MAXINT, MININT
  - Maximum precision of a floating point value
  - Allocation limitation (memory, hard drive space, network bandwidth, etc.)
  - Undefined values

# BASE, EDGE, AND CORNER CASES

- **Base case** – An element in an equivalence class that is not around a boundary (interior value), OR an expected use case.
- **Edge case** – An element in an equivalence class that is next to a boundary (boundary value), OR an unexpected use case.
- **Corner case (or pathological case)** – A case which can only occur outside of normal operating parameters, or a combination of multiple edge cases.



# BLACK-, WHITE, AND GREY-BOX TESTING

- **Black-box testing:** Testing with no knowledge of the interior structure or code of the application. Tests are often performed from the user's perspective, looking at the system as a whole.
- **White-box testing:** Testing with explicit knowledge of the interior structure and codebase, and directly testing that code. Tests are often at a lower level (e.g., testing individual methods or classes)
- **Grey-box testing:** Testing with knowledge of the interior structure and codebase of the system under test, but not directly testing the code. Tests are similar to black-box tests, but are informed by the tester's knowledge of the codebase.

# BLACK-BOX TESTING EXAMPLES

- Accessing a website, using a browser, to look for flaws
- Running a script against an API endpoint
- Checking to see that changing fonts in a word processor shows the correct font

# WHITE-BOX TESTING EXAMPLES

- Testing that a function returns the correct result
- Testing that instantiating an object creates a valid object
- Checking that there are no unused variables in a method

# GREY-BOX TESTING EXAMPLES

- Reviewing code, and noticing that bubble sort is used. Then write a user-facing test involving a large input size.
- Reviewing code and noticing an off-by-one error. Then write a user-facing test which checks that boundary value.

# STATIC VS DYNAMIC TESTING

- Dynamic testing = code is executed (at least some of it)
- Static testing = code is not executed

# DYNAMIC TESTING

- If you're thinking about testing, this is probably what you are thinking about.
  - Code is executed under certain circumstances (e.g. input values, environment variables, etc.)
  - **Observed results** are then compared with **expected results**
- The majority of the class will consists of dynamic testing
- Much more commonly used in industry

# STATIC TESTING

- The code is reviewed by a person or external program, without being executed
- Examples:
  - Code walkthroughs and reviews
  - Requirements analysis
  - Source Code Analysis
    - Linting
    - Model checking
    - Complexity analysis
    - Code coverage
    - Finite state analysis
    - ... COMPILING!