Recitation 8

Brandon Rozek
rozekb@rpi.edu

Rensselaer Polytechnic Institute, Troy, NY, USA

March 2022

Outline

Two Things:

- Testing
- Equality

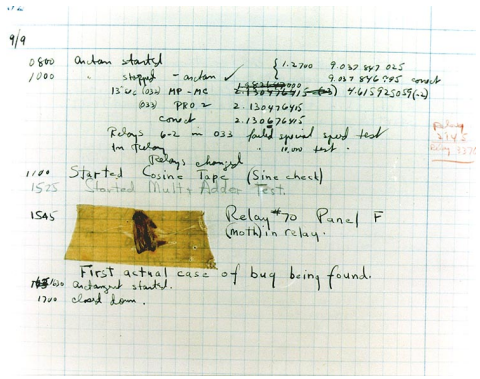
Impact of Bugs

In 2002, NIST reported that software bugs cost the US economy $59.5 billion anually. Software bugs can not only cause catastrophic failures but also provide entryways for hackers.

- February 2022, Mazda owners in the Seattle area who tuned to NPR had their infotainment system bricked because the album art did not contain a file extension.

What is a bug?

- A *software bug* is an error or fault that causes some sort of incorrect, unexpected behavior.
- The term existed for a long time before software, but it is often attributed to the time that Grace Hopper's team found a moth in their computer tape program.



Most Common Bugs

- Memory
  - Null Dereferencing, Buffer Overflow, Unwanted Aliasing
- Arithmetic
  - Division by Zero, Arithmetic (over/under)-flow, Precision
- Parallel Computation
  - Deadlocks and Race Conditions
- Logic
  - Off-by-one and Undecidable / Infinite Looping

Testing

- *Software testing* examines the behavior of software under different conditions for validation and verification.
- Testing is one tool used to expose the presence of bugs but not prove their absence.

Scopes of Testing

- Unit Testing
  - Verify the functionality of a specific section of code.
- Integration Testing
  - Verify the interfaces between components.
  - Make sure NPR doesn't brick your infotainment system!
- System/Acceptance Testing
  - Verify the entire system to see if it meets overall requirements.
  - Am I getting what I paid for?

Test Cases

A test case takes some initial state and input and verifies that the output matches the expected output.

```
public class MyTests {
    @Test
    public void ensureUniqueIDs() {
        // Initialize
        Students psoft = new Students();
        Student sally = psoft.newStudent();
        Student suzy = psoft.newStudent();

        assertNotEquals(sally.userID, suzy.userID);
    }
}
```

Testing Strategies

We will review two testing strategies:

- Black Box Testing
- White Box Testing

Black Box Testing

Otherwise known as *specification-based* testing, this method of testing examines the functionality of an application without looking at its implementation but only its specification.

Common Techniques Include:

- **Equivalence Partitioning**
- **Boundary-Value Analysis**
- Causal Maps
- Error Guessing
- State Transition Table

Selective Testing

- Out of practicality we can't test everything. (Budget Constraints)
- In fact it may be impossible, a method may take an infinite variety of inputs.
- Therefore, we will favor techniques that reduces the input space we need to consider.

Equivalence Partitioning

- Divide the input data into partitions of equivalent data from which test cases can be applied.
- For a method with precondition $0 \leq x \leq 12$ we will have three partitions.
  - $x < 0$, $0 \leq x \leq 12$, $x > 12$
- We can consider subtypes as different partitions as well. For example, a method that takes an `undergraduate` object may have four partitions:
  - `Freshman`, `Sophomore`, `Junior`, `Senior`.

Practice: Equivalence Partitioning

```
/*
@param x an integer
@requires x > 0
@modifies none
@effects x % 3 == 0 => 10 && x % 3 == 1 => 5 && x
    % 3 == 2 => 8
@return an integer
*/
public int fancyfunc(int x);
```

Boundary Value Analysis

- Include tests which includes representatives of a boundary range.
- This technique is applicable for input data that is *ordered*.
- For the example $0 \leq x \leq 12$ the boundary values are $x = 0$ and $x = 12$.

Practice: Boundary Value Analysis

- What are the boundary values for `x is an int`?

Common Boundary Cases

- Arithmetic
  - `Integer.MIN_VALUE`, `Integer.MAX_VALUE`, Zero, Negative, Positive
- Objects
  - Null, Circular List, Aliasing

Practice: Boundary Value Analysis

The following are true:

$$A < B < C < D < E$$

Precondition: $x > D \;||\; x < B$

What are the boundaries?

White Box Testing

Opposite of Black Box Testing, *white box testing* is where we test with knowledge of the code base.

Common techniques include:

- **Control Flow Testing**
- **Code Coverage**
- Data Flow Testing

Control Flow Based Testing

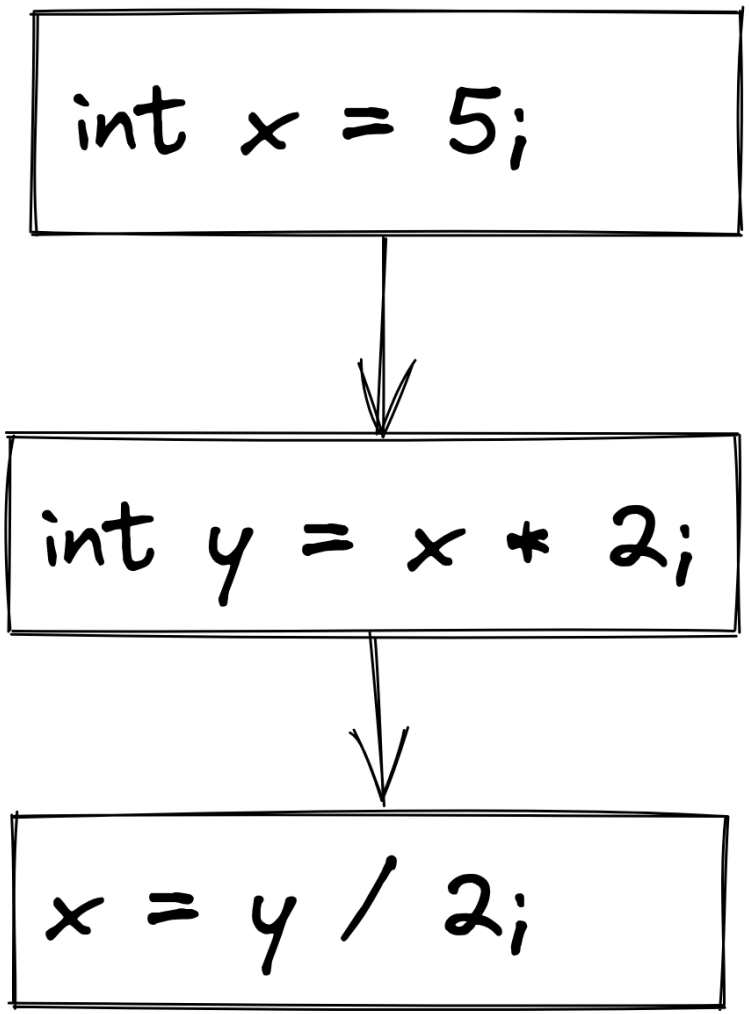
- Control Flow Based Testing is the idea of extracting a *control flow graph* (CFG) and creating test suits that provides *coverage* for all of the nodes in the graph.

Control Flow Graph

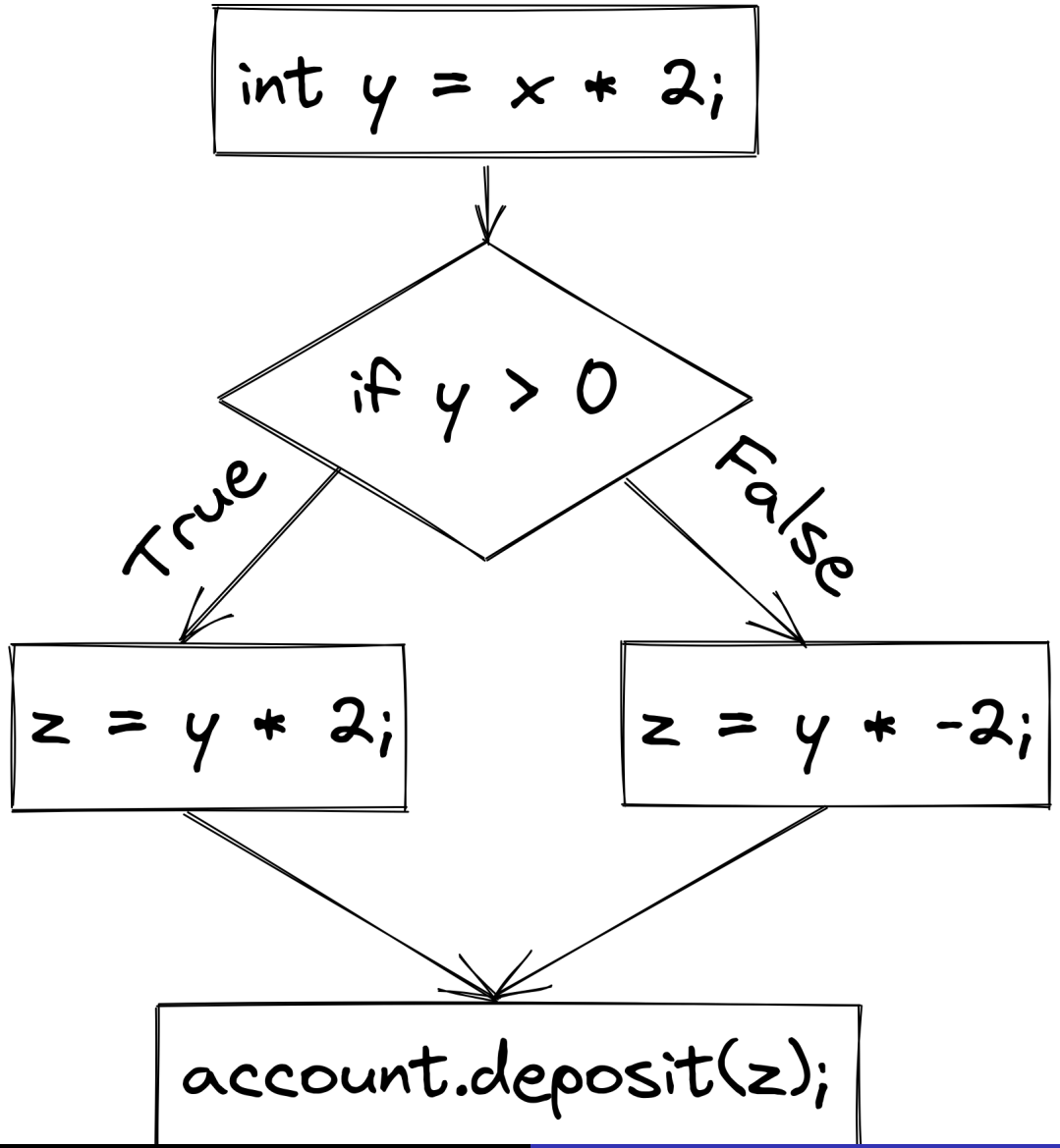
We can break a CFG into:

- Sequence of Statements
- Conditional branch if some condition is met
- Conditional loop until some condition is met
- Executing a set of distant statements, after which the flow of control returns (methods, subroutines, etc.)

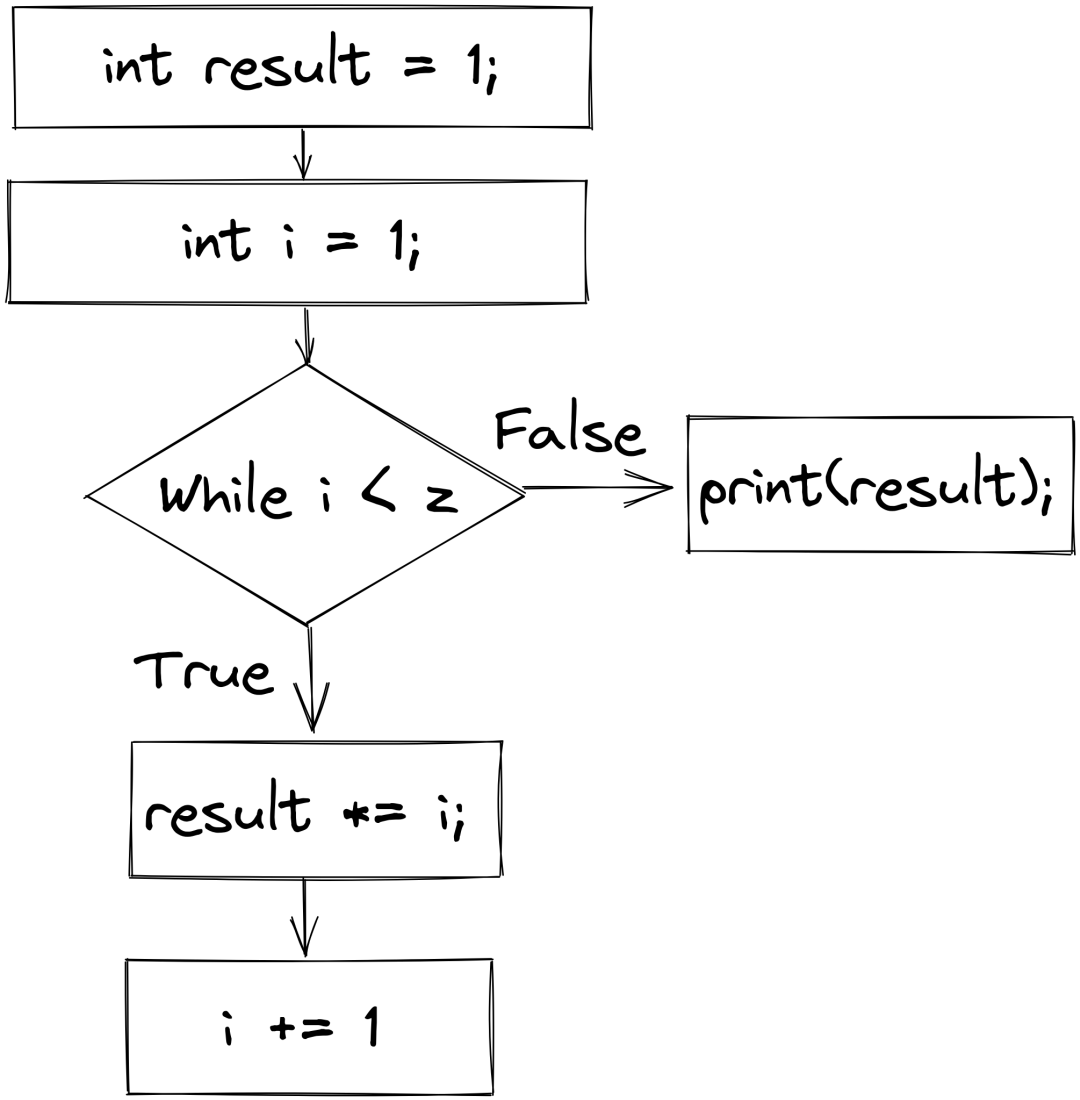
Sequence of Statements



Conditional Branch



Conditional Loop



Reference Equality

In Java, == tests for reference equality. That is, do the two variable names point to the same object in memory?

Equivalence Relation

A binary relation R is said to be an equivalence relation if and only if it is reflexive, symmetric, and transitive. That is:

- Reflexive: $\forall x : xRx$
- Symmetric: $\forall x, y : xRy \iff yRx$
- Transitive: $\forall x, y, z : xRy \wedge yRz \implies xRz$

Value Equality

In Java, to check for value equality, you'll need to override the classes `.equals(Object other)` method.

Important notes to make sure of:

- You **override** not overload the equals method.
- The input argument is of type `Object`
- The access modifier is `public`
- We satisfy the equivalence relation properties

Value Equality Code Example

```
public class Person {
    private String name;
    @Override
    public boolean equals(Object obj) {
        if (!(obj isinstanceOf Person)) {
            return false;
        }
        Person p2 = (Person) obj;
        return this.name == p2.name;
    }
}
```

Hash Function

- A *hash function* maps data of arbitrary size to a fixed-size value.
- Often used for data structures that feature fast lookups.

Desirable Hash Function Properties

- Uniformity: The probability of an object taking some value is the same as it taking another value in the hash function.
- Efficient: We can't have fast lookups, if the hash takes time to compute.
- Deterministic: Given some input, we will always get the same output.

Hash Codes in Java

If we override `.equals(...)` then we must also override `.hashCode()`.

```
public class Person {
    private String name;
    @Override
    public int hashCode() {
        final int prime = 197;
        int result = 1;
        result = prime * result + name.hashCode();
        return result;
    }
}
```

Any Questions?