SENG 275

SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

TESTING TECHNIQUES- BOUNDARY VALUE ANALYSIS

University of Victoria

Boundary (Edge) testing

- **Off-by-one mistakes** are a common cause for bugs in software systems.
- As developers, we have all made mistakes such as using a "greater than" operator (**>**) where it had to be a "greater than or equal to" operator (**>=**).
- Interestingly, programs with such a bug tend to **work well for most of the provided inputs.**
- They fail, however, when the input is "**near the boundary of condition".**
- Now, we explore boundary testing techniques.

Boundaries in between classes/partitions

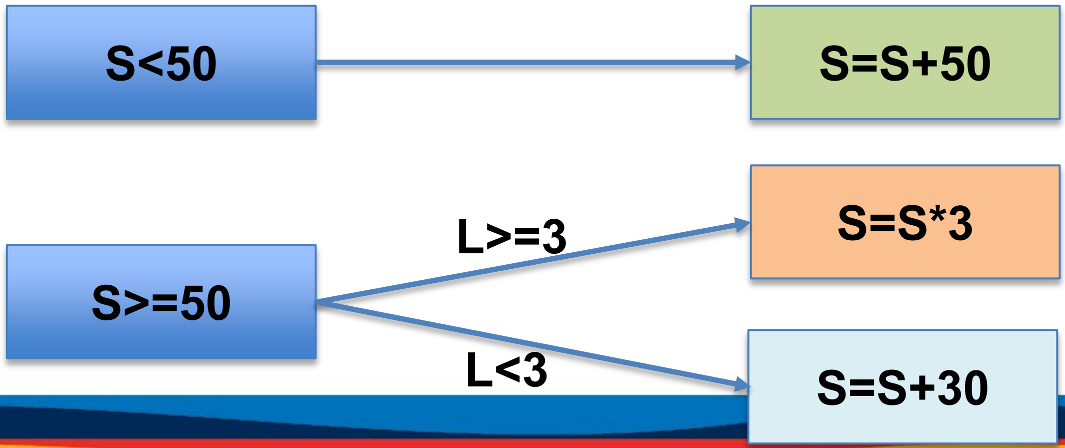
- Previously, we studied specification-based techniques and, more specifically, we understood the concept of classes/partitions.
- When we devise classes, these have "**close boundaries" with the other classes.**
- In other words, if we keep performing small changes to an input that belongs to some partition (e.g., by adding +1 to it), at some point this input will belong to another class.
- The **precise point** where the input changes from **one class to another** is what we call a **boundary**.
- And this is precisely what boundary testing is about: to make the program **behave correctly when inputs are near a boundary**.

University of Victoria

Requirement: Calculating the number of points of a player

In a game, given the score of a player and the number of **remaining lives** of the player, the program does the following:

- If the player's score is **below 50**, then it always **adds 50 points** on top of the current points.
- If the player's score is **greater than or equals to 50**, then:
  - –if the number of remaining **lives is greater than or equal to 3**, it **triples the score** of the player.
  - –otherwise, it **adds 30 points** on top of the current points.

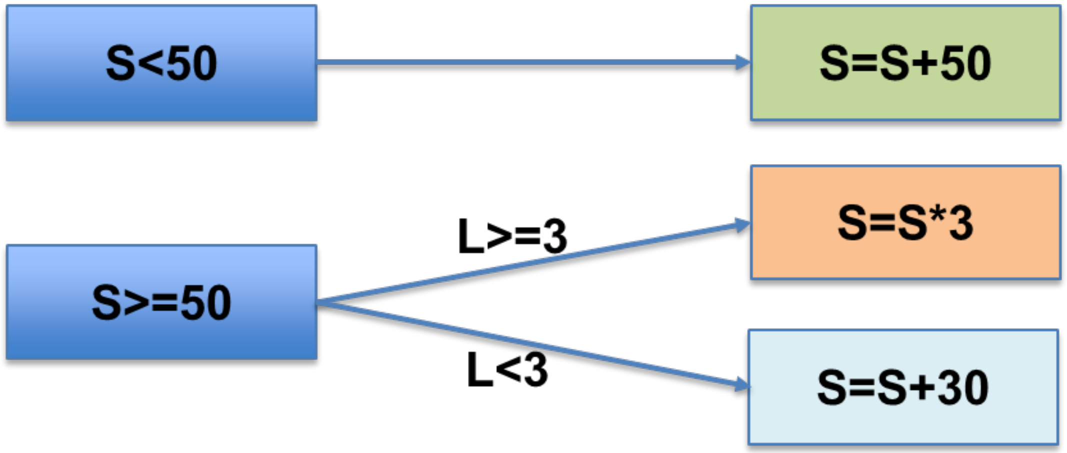


University of Victoria

Requirement: Calculating the number of points of a player

Eg:

- **score:14 -> score+50=64**
- **score: 65 -> lives=3 or 4 -> score*3=195**
- **score: 65 -> lives=2 -> score+30=95**



University of Victoria

Create input domain Partitions (Equivalent Partitioning)

When devising the partitions to test this method, a tester might come up with the following partitions:

Test input:[score, remaining lives]

1. Less score(E.Partition 1): Score < 50 , Test input:[30,5],

2. More score but less lives(E.Partition 2): Score >= 50 and remaining lives < 3, Test input:[300,1],

3. More score and many lives(E.Partition 3): Score >= 50 and remaining lives >= 3, Test input:[500,10],

Create junit test classes for these 3 partitions.

One possible implementation

```
public class PlayerPoints {


public int totalPoints(int currentPoints, int remainingLives) {
    if(currentPoints < 50)
        return currentPoints+50;
   else
        return remainingLives < 3 ? currentPoints+30 :
currentPoints*3;
  }
}
```

University of Victoria

```
public class PlayerPointsTest {

  private final PlayerPoints pp = new PlayerPoints();

  @Test
  void lessPoints() {
    assertEquals(30+50, pp.totalPoints(30, 5));
  }

  @Test
  void manyPointsButLittleLives() {
    assertEquals(300+30, pp.totalPoints(300, 1));
  }

  @Test
  void manyPointsAndManyLives() {
    assertEquals(500*3, pp.totalPoints(500, 10));
  }
}
```

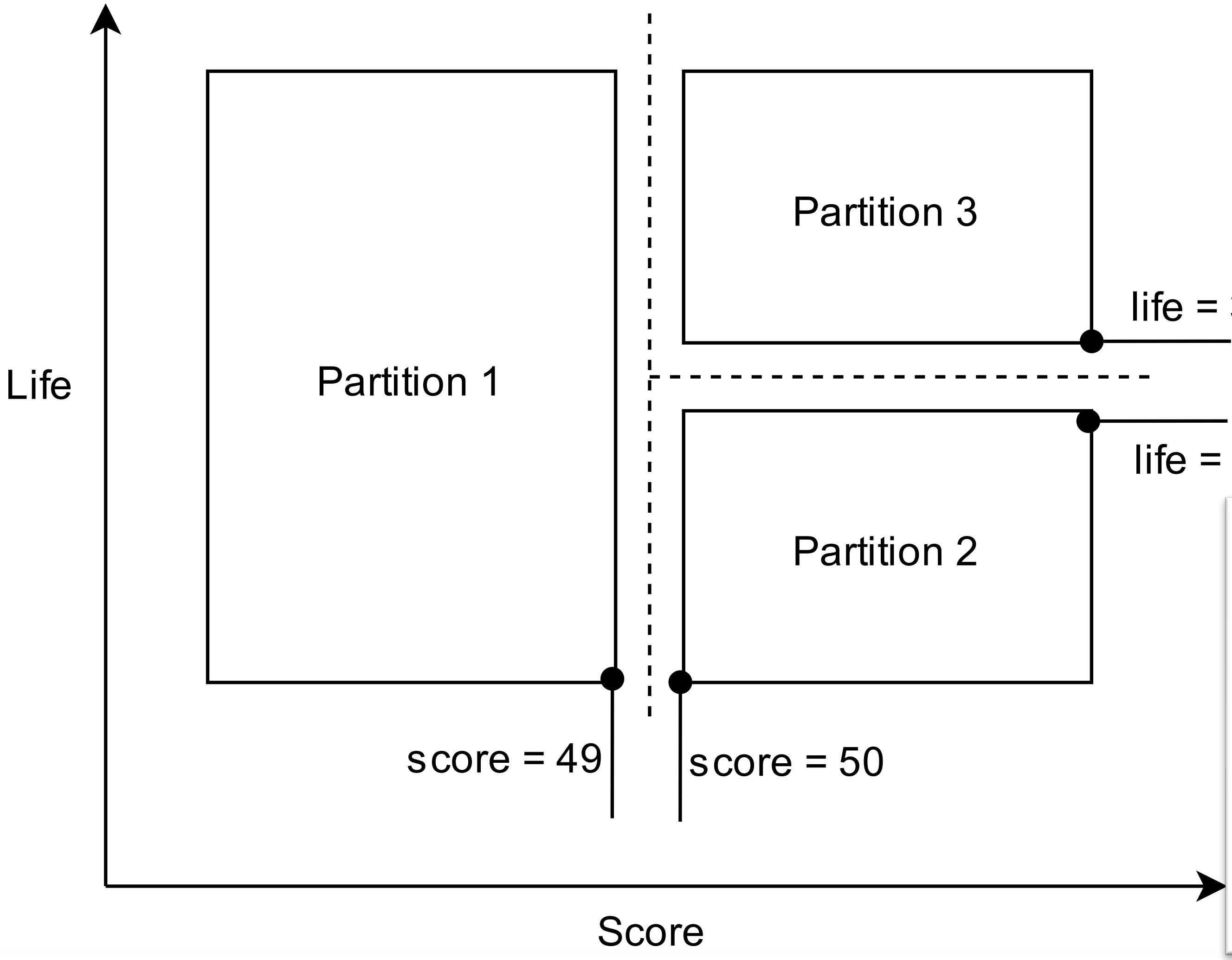
Adding boundary conditions

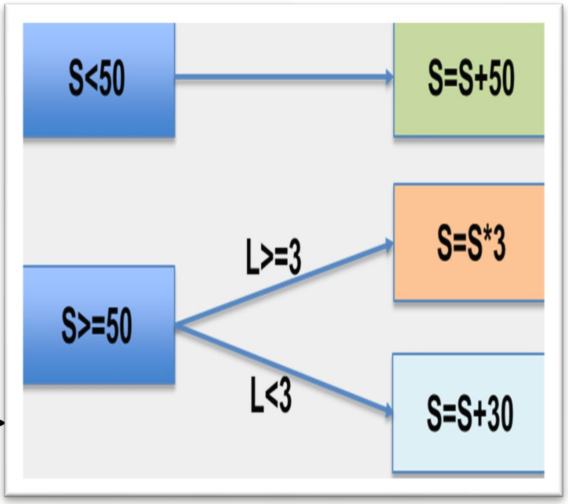
- However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain.
- **Boundary 1:** When the score is **strictly smaller than 50**, it belongs to **partition 1**. If the score is **greater than or equal to 50**, it belongs to **partitions 2 and 3**. Therefore, we observe the following boundary: **when the score changes from 49 to 50, the partition it belongs to also changes** (let us call this **test B1).**
- **Boundary 2:** Given a score that is greater than or equal to 50, we observe that if the number **of remaining lives is smaller than 3, it belongs to partition 2**; otherwise, it belongs to **partition 3**. Thus, we just identified another boundary there (let us call this **test B2**).

University of Victoria

We can visualize these partitions with their boundaries in a diagram.

Boundary value test inputs (only on points)
[49,5]
[50,5]
[500,2]
[500,3]





Add boundary value test cases to the test suite

- In our example, the tester would then devise and automate test cases B1 and B2.
- Given that a boundary is composed of two different input values, note that each boundary will require *at least* two test cases:

For B1:

- B1.1 = input={score=49, remaining lives=5}, output={99}
- B1.2 = input={score=50, remaining lives=5}, output={150}

For B2:

- B2.1 = input={score 500, remaining lives=3}, output={1500}
- B2.2 = input={score 500, remaining lives=2}, output={530}

Add boundary value test cases to the test suite

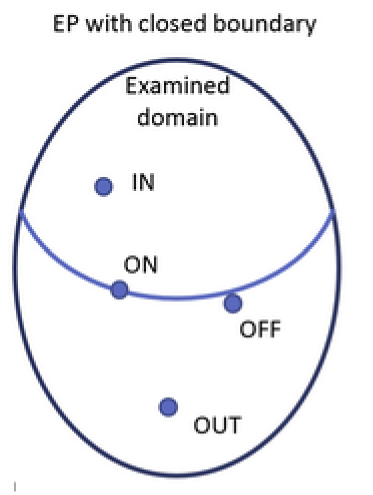
```
@Test
void betweenLessAndManyPoints() {
  assertEquals(49+50, pp.totalPoints(49, 5));
  assertEquals(50*3, pp.totalPoints(50, 5));
}

@Test
void betweenLessAndManyLives() {
  assertEquals(500*3, pp.totalPoints(500, 3));
  assertEquals(500+30, pp.totalPoints(500, 2));
}
```

University of Victoria

- You might have noticed that, for B1, in case of score < 50, remaining lives makes no difference. However, for score >= 50, remaining lives do make a difference, as the output can vary according to its value. And for the B1.2 test case, we chose remaining lives = 5, which makes the condition true. You might be wondering whether you also need to devise another test case, B1.3, where the remaining lives condition would be exercised as false.
- If you are looking to test all possible combinations, then the answer is yes. However, in longer conditions, full of boundaries, the number of combinations might be too high, making it unfeasible for the developer to test them all. Later in this course, we will learn how to choose values for the "boundaries that we do not care about".

ON AND OFF POINTS



A data point on the closed boundary is called an ON point.
A data point outside a closed boundary and "closest" to the ON point is called an OFF point.
A data point inside the examined domain that differs from the ON point and the neighbour of the ON point is called an IN point.
A data point outside the boundary of the examined domain and not being an OFF point is called an OUT point

Ref: https://test-design.org/general-predicate-testing/

On and off points

- Given some initial intuition on how to analyze boundaries, let us define some terminology:
- **On-point:** The on-point is the value that is **exactly on the boundary**. This is the value we see in the condition itself.
- **Off-point**: The off-point is the value that is **closest to the boundary** and that **flips the condition**. If the on-point makes the condition true, the off point makes it false and vice versa. Note that when dealing with **equalities or inequalities** (e.g. $x = 6$ or $x \neq 6$), there are **two** off-points; one in each direction.
- **In-points**: In-points are all the values that make the **condition true**.
- **Out-points**: Out-points are all the values that make the **condition false**.

Example: on, off, in, out points

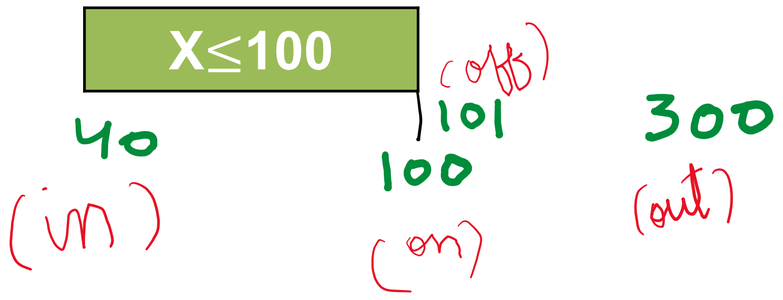
X<100

(in) 40 (off) 99 100 (on) (out) 300

- Suppose we have a program that **adds shipping costs** when the **total price is below 100**. The condition used in the program is **x < 100**.
- The **on-point is 100**, as that is the value that is **precisely in the condition.**
- The **on-point makes the condition false** in this case(100 is not smaller than 100), so the **off-point** should be the closest number that makes the condition **true**. This will be **99**, as 99 < 100 is true.
- The **in-points** are the values which are **smaller than or equal to 99**. For example, **37, 42, 56**.
- The **out-points** are all values which are **larger than or equal to 100**. For example, **325, 1254, 101.**

University of Victoria

Different condition: x ≤100



- Let us now study a similar but slightly different condition: x ≤100 (the only difference is that, in this one, we use "less than or equal to"):
- The **on-point** is still **100**: this is the value that is precisely in the condition.
- **The condition is evaluated as true for the on-point in this case.**
- So, the **off-point** should be the closest number to the on-point, but making the condition **false**.
- The off-point is thus **101**.

How many Test cases ?

There is *no perfect answer* here.

We suggest:

- If the number of test cases is indeed too high, and it is just too expensive to do them all, prioritization is important, and we suggest testers to indeed **focus on the boundaries**.
- Far away in/out points are sometimes easier to be seen or comprehended by a tester who is still learning about the system under test and exploring its boundaries (*exploratory testing*). Deciding whether to perform such a test is thus a decision that a tester should take, taking the **costs** into account.

Lean approach to the number of Test cases

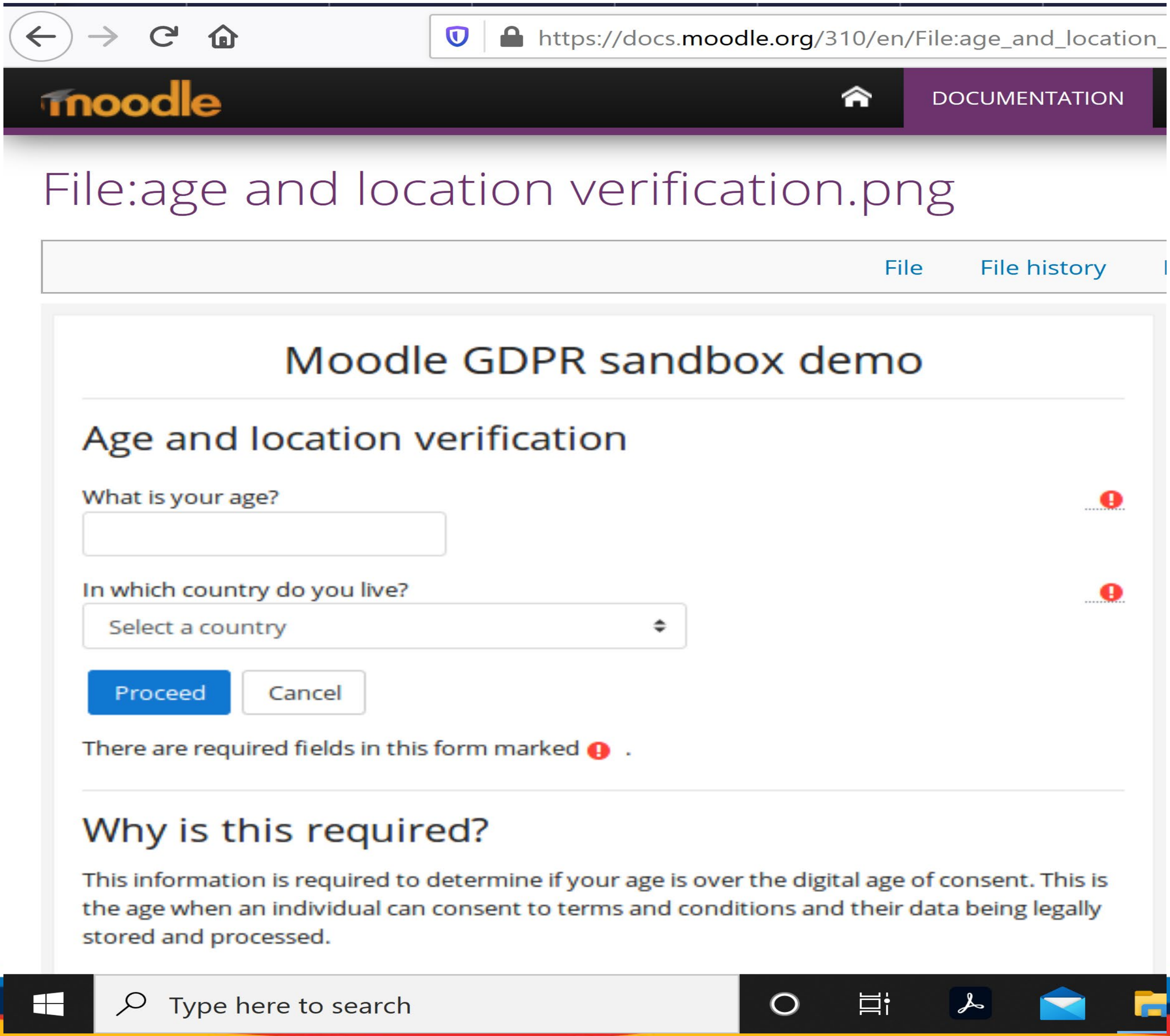
- As a tester, you devise test cases (lean approach) for these different points:
  - a test case for the **on-point**,
  - a test case for the **off-point**,
  - a test case for a **single in-point** (as all in-points belong to the same equivalence partition), and
  - a test case for a **single out-point** (as all out-points also belong to the same equivalence partition).
- Note that *on* and *off* points are also *in* or *out points*.
- Therefore, tests that focus only on the *on* and *off* points would also be testing *in* and *out* points. In fact, **some authors argue that testing boundaries is enough**. Moreover, a test that exercises an in-point that is far away from the boundary might not have a strong fault detection capability. Why would we need them?

Equivalent classes and boundary analysis altogether- *domain testing*.

The following strategy should be applied for domain testing:

1. We read the **requirement**
2. We identify the **input and output variables** in play, together with their **types**, and their **ranges**.
3. We identify the **dependencies** (or independence) among input variables, and how input variables influence the output variable.
4. We perform **equivalent class analysis** (valid and invalid classes).
5. We explore the **boundaries** of these classes.
6. We think of a **strategy** to derive test cases, focusing on **minimizing the costs while maximizing fault detection** capability.
7. We generate a **set of test cases** that should be executed against the system under test.

Age can be between 18-60



File:age and location verification.png

Moodle GDPR sandbox demo

Age and location verification

What is your age?

In which country do you live?

Select a country

Proceed Cancel

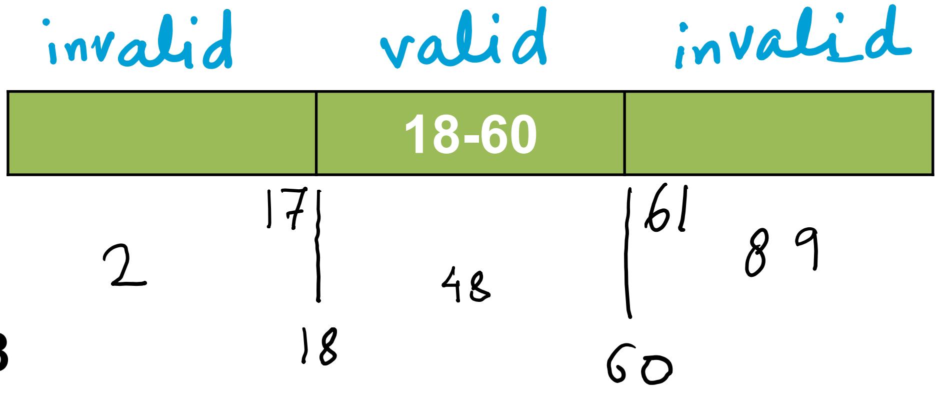
There are required fields in this form marked ❗ .

Why is this required?

This information is required to determine if your age is over the digital age of consent. This is the age when an individual can consent to terms and conditions and their data being legally stored and processed.

University of Victoria

devise test cases



- Boundary1 (18)
  - –a test case for the **on-point:18**
  - –a test case for the **off-point: 17 (shd. Make cond. False if on-point is making cond. True)**
  - –a test case for a **single in-point: 48**
  - –a test case for a **single out-point: 2**
- Boundary2 (60)
  - –a test case for the **on-point:60**
  - –a test case for the **off-point**: **61**
  - –a test case for a **single in-point: 37**
  - –a test case for a **single out-point: 89**

Java program to check age

```
public class Age {
 public boolean checkAge(int a)
 {
 if (a>=18 && a<=60)
 return true;
 else
 return false;
 }
```

University of Victoria

```
public class AgeTest {
  Age age=new Age();
  @Test
  public void test1()
  {  boolean b=age.checkAge(18);
     assertTrue(b);     }
  @Test
  public void test2()
  {  boolean b=age.checkAge(17);
     assertTrue(b);     }
  @Test
  public void test3()
  {  boolean b=age.checkAge(48);
     assertFalse(b);   }

@Test
  public void test4()
  {  boolean b=age.checkAge(2);
     assertTrue(b);     }
  @Test
  public void test5()
  {  boolean b=age.checkAge(60);
     assertTrue(b);     }
  @Test
  public void test6()
  {  boolean b=age.checkAge(61);
     assertFalse(b);      }
@Test
  public void test5()
  {  boolean b=age.checkAge(37);
     assertTrue(b);     }
  @Test
  public void test6()
  {  boolean b=age.checkAge(89);
     assertFalse(b);      }
}
```

University of Victoria

Try this !!!!!

- Q2) A game has the following condition: numberOfPoints <= 570. Perform boundary analysis on the condition. What are the on- and off-point of the condition? Also give an example for both an in-point and an out-point. (4M)