

Void Safety



EECS3311: Software Design
Fall 2017

CHEN-WEI WANG

Java Program: Example 1

```
1 class Point {
2     double x;
3     double y;
4     Point(double x, double y) {
5         this.x = x;
6         this.y = y;
7     }
}
```

```
1 class PointCollector {
2     ArrayList<Point> points;
3     PointCollector() { }
4     void addPoint(Point p) {
5         points.add(p); }
6     Point getPointAt(int i) {
7         return points.get(i); } }
}
```

The above Java code **compiles**. But anything wrong?

```
1 @Test
2 public void test1() {
3     PointCollector pc = new PointCollector();
4     pc.addPoint(new Point(3, 4));
5     Point p = pc.getPointAt(0);
6     assertTrue(p.x == 3 && p.y == 4); }
}
```

L3 calls PointCollector constructor not initializing points.
∴ **NullPointerException** when **L4** calls **L5** of PointCollector.

Java Program: Example 2

```
1 class Point {
2     double x;
3     double y;
4     Point(double x, double y) {
5         this.x = x;
6         this.y = y;
7     }
}
```

```
1 class PointCollector {
2     ArrayList<Point> points;
3     PointCollector() {
4         points = new ArrayList<>(); }
5     void addPoint(Point p) {
6         points.add(p); }
7     Point getPointAt(int i) {
8         return points.get(i); } }
}
```

```
1 @Test
2 public void test2() {
3     PointCollector pc = new PointCollector();
4     Point p = null;
5     pc.addPoint(p);
6     p = pc.getPointAt(0);
7     assertTrue(p.x == 3 && p.y == 4); }
}
```

The above Java code **compiles**. But anything wrong?

L5 adds `p` (which stores `null`).

\therefore **NullPointerException** when **L7** calls `p.x`.

Java Program: Example 3

```
1 class Point {
2     double x;
3     double y;
4     Point(double x, double y) {
5         this.x = x;
6         this.y = y;
7     }
}
```

```
1 class PointCollector {
2     ArrayList<Point> points;
3     PointCollector() {
4         points = new ArrayList<>(); }
5     void addPoint(Point p) {
6         points.add(p); }
7     Point getPointAt(int i) {
8         return points.get(i); } }
}
```

```
1 public void test3() {
2     PointCollector pc = new PointCollector();
3     Scanner input = new Scanner(System.in);
4     System.out.println("Enter an integer:");
5     int i = input.nextInt();
6     if(i < 0) { pc = null; }
7     pc.addPoint(new Point(3, 4));
8     assertTrue(pc.getPointAt(0).x == 3 && pc.getPointAt(0).y == 4);
9 }
```

The above Java code **compiles**. But anything wrong?

NullPointerException when user's input at **L5** is non-positive.

Limitation of Java's Type System

- A program that compiles does not guarantee that it is free from **NullPointerException** :
 - Uninitialized attributes (in constructors).
 - Passing **nullable** variable as a method argument.
 - Calling methods on **nullable** local variables.
- The notion of `Null` references was back in 1965 in ALGO W.
- Tony Hoare (inventor of Quick Sort), introduced this notion of `Null` references “simply because *it was so easy to implement*”.
- But he later considers it as his “**billion-dollar mistake**”.
 - When your program manipulates reference/object variables whose types include the legitimate value of `Null` or `Void`, then there is always a possibility of having a **NullPointerException** .
 - For undisciplined programmers, this means the final software product **crashes** often!

Eiffel's Type System for Void Safety

- By default, a reference variable is *non-detachable*.
 e.g., `acc: ACCOUNT` means that `acc` is always *attached* to some valid `ACCOUNT` point.
- VOID** is an illegal value for *non-detachable* variables.
 ⇒ Scenarios that might make a reference variable *detached* are considered as *compile-time errors*:
 - Variables can not be assigned to `Void` directly.
 - Non-detachable* variables can only be re-assigned to *non-detachable* variables.

e.g., `acc2: ACCOUNT` ⇒ `acc := acc2` *compilable*

e.g., `acc3: detachable ACCOUNT` ⇒ `acc := acc3` *non-compilable*

- Creating variables (e.g., `create acc.make`) *compilable*
- Non-detachable* attribute not explicitly initialized (via creation or assignment) in all constructors is *non-compilable*.

Eiffel Program: Example 1

```
1 class
2   POINT
3 create
4   make
5 feature
6   x: REAL
7   y: REAL
8 feature
9   make (nx: REAL; ny: REAL)
10  do x := nx
11     y := ny
12  end
13 end
```

```
1 class
2   POINT_COLLECTOR_1
3 create
4   make
5 feature
6   points: LINKED_LIST[POINT]
7 feature
8   make do end
9   add_point (p: POINT)
10  do points.extend (p) end
11   get_point_at (i: INTEGER): POINT
12  do Result := points [i] end
13 end
```

- Above code is semantically equivalent to Example 1 Java code.
- Eiffel compiler won't allow you to run it.
 - ∴ L8 does **non compile**
 - ∴ It is **void safe** [Possibility of **NullPointerException** ruled out]

Eiffel Program: Example 2

```

1  class
2    POINT
3  create
4    make
5  feature
6    x: REAL
7    y: REAL
8  feature
9    make (nx: REAL; ny: REAL)
10   do x := nx
11     y := ny
12   end
13 end
  
```

```

1  class
2    POINT_COLLECTOR_2
3  create
4    make
5  feature
6    points: LINKED_LIST[POINT]
7  feature
8    make do create points.make end
9    add_point (p: POINT)
10   do points.extend (p) end
11   get_point_at (i: INTEGER): POINT
12   do Result := points [i] end
13 end
  
```

```

1  test_2: BOOLEAN
2  local
3    pc: POINT_COLLECTOR_2 ; p: POINT
4  do
5    create pc.make
6    p := Void
7    p.add_point (p)
8    p := pc.get_point_at (0)
9    Result := p.x = 3 and p.y = 4
10 end
  
```

- Above code is semantically equivalent to Example 2 Java code.
 L7 does **non compile** ∴ pc might be void. [void safe]

Eiffel Program: Example 3

```

1  class
2    POINT
3  create
4    make
5  feature
6    x: REAL
7    y: REAL
8  feature
9    make (nx: REAL; ny: REAL)
10   do x := nx
11     y := ny
12   end
13 end

```

```

1  class
2    POINT_COLLECTOR_2
3  create
4    make
5  feature
6    points: LINKED_LIST[POINT]
7  feature
8    make do create points.make end
9    add_point (p: POINT)
10   do points.extend (p) end
11   get_point_at (i: INTEGER): POINT
12   do Result := points [i] end
13 end

```

```

1  test_3: BOOLEAN
2  local pc: POINT_COLLECTOR_2 ; p: POINT ; i: INTEGER
3  do create pc.make
4    io.print ("Enter an integer:%N")
5    io.read_integer
6    if io.last_integer < 0 then pc := Void end
7    pc.add_point (create {POINT}.make (3, 4))
8    p := pc.get_point_at (0)
9    Result := p.x = 3 and p.y = 4
10 end

```

- Above code is semantically equivalent to Example 3 Java code. L7 and L8 do **non compile** ∴ pc might be void. [void safe]

Lessons from Void Safety

- It is much more costly to recover from *crashing* programs (due to `NullPointerException`) than to fix *uncompilable* programs.
e.g., You'd rather have a *void-safe design* for an airplane, rather than hoping that the plane won't crash after taking off.
- If you are used to the standard by which Eiffel compiler checks your code for *void safety*, then you are most likely to write Java/C/C++/C#/Python code that is *void-safe* (i.e., free from *NullPointerExceptions*).

Beyond this lecture...

- Tutorial Series on Void Safety by Bertrand Meyer (inventor of Eiffel):
 - The End of Null Pointer Dereferencing
 - The Object Test
 - The Type Rules
 - Final Rules
- Null Pointer as a Billion-Dollar Mistake by Tony Hoare
- More notes on void safety

Index (1)

Java Program: Example 1

Java Program: Example 2

Java Program: Example 3

Limitation of Java's Type System

Eiffel's Type System for Void Safety

Eiffel Program: Example 1

Eiffel Program: Example 2

Eiffel Program: Example 3

Lessons from Void Safety

Beyond this lecture...