Testing Driven Development

Advanced Programming

Test Driven Development (TDD)

- Practice for writing unit tests and production code concurrently and at a very fine level of granularity
- Programmers
  - first write a small portion of a unit test, and
  - then they write just enough production code to make that unit test compile and execute

Test Driven Development (TDD)

- This cycle lasts somewhere between 30 seconds and five minutes. Rarely does it grow to ten minutes.
- In each cycle, the tests come first.
- Once a unit test is done, the developer goes on to the next test until they run out of tests for the task they are currently working on

Example - TDD

- TextFormatter: A text formatter that take arbitrary strings and horizontally center them in a page
- Few issues:
- What are the methods:
  - setLineWidth()
  - centerLine()
- What is a Line?
- Can I use String?

What to test

- First understand the entities to test

String and StringBuffer

- String is immutable; that is, it cannot be modified once created
- If a String object is modified, a new String was actually created and the old one was thrown away.

Example

```
String badlyCutText = "     Java is great.     ";
System.out.println(badlyCutText);
badlyCutText.trim(); //attempt to modify the string
System.out.println(badlyCutText);
```

Output

```
    Java is great.
    Java is great.
```

String

- The String.trim() method returns the string with leading and trailing whitespace removed
- The trim() method call does not modify the original object
  - It creates a new trimmed String object and then throws it away
  - Thus, we we print the string we get the original String object

String

- Once a String object is created, it can not be modified, takes up memory until garbage collection

To trim the original String

```
String badlyCutText = "      Java is great.      ";

System.out.println(badlyCutText);

badlyCutText =badlyCutText.trim();

System.out.println(badlyCutText);
```

Output

```
      Java is great.

Java is great.
```

Using StringBuilder/StringBuffer

- With immutable objects, we need to store the modified object in a new reference variable

Raw concatenation

```
public String convertToString(Collection<String> words) {
  String str = "";
  // Loops through every element in words collection
  for (String word : words) {
     str = str + word + " ";
  }
  return str;
}
```

Raw concatenation

- On the + operation a new String object is created at each iteration.
- Suppose words contains the elements [“Foo”, “Bar”, “Bam”, “Baz”]. The method creates eleven Strings: “”, “Foo”, “ ”, “Foo ” , “Foo Bar”, “ ”, “Foo Bar ”, “Foo Bar Bam”, “ ”, “Foo Bar Bam ”, “Foo Bar Bam Baz”
- Even though only the last one is actually useful.
- Memory is only cleaned by the garbage collector

Raw concatenation

- To avoid unnecessary memory use like this, use the StringBuilder class
  - Only one StringBuilder object is created.
  - Also because object creation is time consuming, using StringBuilder produces much faster code
- It provides similar functionality to Strings, but stores its data in a mutable way

Concatenation with StringBuilder

```
public String convertToString(Collection<String> words) {
  StringBuilder buffer = new StringBuilder();
  // Loops through every element in words collection
  for (String word : words) {
      buffer.append(word);
      buffer.append(" ");
  }
  return buffer.toString();
}
```

StringBuilder / StringBuffer

- As StringBuilder is not thread safe you cannot use it in more than one thread.
- Use StringBuffer instead, which does the same and is thread safe
  - StringBuffers are thread-safe: they have synchronized methods to control access so that only one thread can access a StringBuffer object's synchronized code at a time.

StringBuffer

- However, as StringBuffer is slower, only use StringBuffer in a multi-thread environment
- Note: only StringBuffer exists before Java 5

StringBuilder

- If you are working in a single-threaded environment, using StringBuilder instead of StringBuffer may result in increased performance.
- So, prefer StringBuilder because,
  - Small performance gain.
  - StringBuilder is a 1:1 drop-in replacement for the StringBuffer class.
  - StringBuilder is not thread synchronized and therefore performs better on most implementations of Java

First we write the test	*Then we write the production code*

public void testCenterLine(){
  Formatter f = new Formatter();
}
does not compile
	class Formatter{ }
	
	compiles and passes

public void testCenterLine(){
  Formatter f = new Formatter();
  f.setLineWidth(10);
  assertEquals(" word ", f.center("word"));
 	
}
 	
does not compile
	class Formatter{
	  public void setLineWidth(int width) { }
	  public String center(String line) {
	  return "";
	  }
	}
	
	compiles and fails

	import java.util.Arrays;
	public class Formatter {
	  private int width;
	  private char spaces[];
	public void setLineWidth(int width) {
	  this.width = width;
	  spaces = new char[width];
	  Arrays.fill(spaces, ' ');
	}
	public String center(String term) {
	  StringBuffer b = new StringBuffer();
	  int padding = width/2 - term.length();
	  b.append(spaces, 0, padding);
	  b.append(term);
	  b.append(spaces, 0, padding);
	  return b.toString();
	}
	}
	
	compiles and unexpectedly fails

	public String center(String term) {
	  StringBuffer b = new StringBuffer();
	  int padding = (width - term.length()) / 2;
	  b.append(spaces, 0, padding);
	  b.append(term);
	  b.append(spaces, 0, padding);
	  return b.toString();
	}
	compiles and passes

public void testCenterLine() {
  Formatter f = new Formatter();
  f.setLineWidth(10);
  assertEquals(" word ", f.center("word"));
}
  public void testOddCenterLine() {
  Formatter f = new Formatter();
  f.setLineWidth(10);
  assertEquals(" hello ", f.center("hello"));
}
compiles and fails
	public String center(String term) {
	  int remainder = 0;
	  StringBuffer b = new StringBuffer();
	  int padding = (width - term.length()) / 2;
	  remainder = term.length() % 2;
	  b.append(spaces, 0, padding);
	  b.append(term);
	  b.append(spaces, 0, padding + **remainder**);
	  return b.toString(); }
	compiles and passes

Exercise

- Extend the previous example by allowing any line length

Exercise

- Extend the example above by allowing terms that are concatenation of word

What are the benefits of TDD?

- Line Test Coverage: If you follow the rules of TDD, then virtually 100% of the lines of code in your production program will be covered by unit tests
  - This does not cover 100% of the paths through the code, but it does make sure that virtually **every line is executed** and tested.

What are the benefits of TDD?

- Test Repeatability. The tests can be run any time you like.
- Documentation. The tests describe your understanding of how the code should behave. They also describe the API. Therefore, the tests are a form of documentation.

What are the benefits of TDD?

- API Design. When you write tests first, you put yourself in the position of a user of your program's API. This can only help you design that API better.
- Reduced Debugging. When you move in the tiny little steps recommended by TDD, it is hardly ever necessary to use the debugger. Debugging time is reduced enormously.