Test-Driven Development (TDD)

- Underlying philosophy
  - Test code interleaved with system code development yields more reliable software.

- Advantages
  - Increases confidence of code correctness.
  - Bugs/mistakes are easier to locate/fix in code immediately after it is written since it still fresh in the programmer’s mind.
    - Code is developed such that classes/methods are testable in isolation.
  - Helps ensure that code satisfies system requirements.
    - The tests serve as system-level documentation.
  - Decreases debugging time.
  - Uncovers design flaws
  - Integrates naturally with stepwise refinement & problem decomposition
    - Allows functionality to be added incrementally
Composing test code
- Test code is developed in a separate test class - automated.
- Objects of the class under test are instantiated in the test class.
- Code which exercises the objects is written.
- Assertion code which expresses the correct behavior of the objects under exercise is written.

Testing
- The goal of tests are to discover errors in your code.
- Writes tests that purposefully attempt to cause code to fail.
- When code it changed, tests on that code that previously passed must be re-executed.

*Program testing can best show the presence of errors but never their absence.*

--Edsger Dijkstra: 1930-2002
**Approach**
- Write a small amount of code, write the test code for it
  - Tests code is small and targeted
- Alternatively, write the test code for each method prior to writing the method.
  - As soon as the method is written it is ready to be tested with the previously written test code.

**Unit Testing**
- Simple accessor/mutator methods can be tested by using them to test construction and other methods, (simple == 1 line of code).

**TDD Development Cycle**
- Code the tests for the next module to be developed
- Code the module so that it passes the tests
- Execute the tests
- Refactor/modify the code related to failed tests (automated process)
- Repeat the TDD development cycle
BlueJ's testing features

- Tools->Preferences... Miscellaneous tab check "Show unit testing tools"

Test Class

- Right-click class, select "Create Test Class"

Test Fixture

- a collection of objects that have already been created, with invoked methods to transform the objects into the state to be tested/checked.
- Can be created interactively or directly by coding

Interactive Test Fixture

- Instantiate object(s) to be tested
- Invoke method(s) to be created, (transform objects to test state)
- Right-click the test class. Select "Object Bench to Test Fixture."
- This records all of the objects on the object bench inside the test class.
  - Adds object(s) instantiation and method invocation code to the test class.
  - Each test that you add to this test class will be run with the current configuration of objects as its starting point.
Test Case Creation
- A "test case" is one particular test to check.
- Includes a statement of what actions to perform, together with a statement of how to check whether those actions had the desired effect.
- First part is accomplished by the test fixture.
- Second part is accomplished by test methods.

Test Method Creation
- Right-click test class, select "Create Test Method...".
- Give test method a meaningful name indicating what it checks.
- Object bench reverts to the stored "test fixture" state for this test class.
- Red "recording" light in BlueJ's window comes on.
  - Any actions taken now are "recorded" as part of the test case.
  - Right-click on objects and invoke method to be tested.
  - Finish test case by clicking "End" button.
JUnit Test Assertions

- **Object State Checks**
  - Edit the test class/method just created.
  - Add assertions to the test method to ensure that the state change has occurred correctly.

- **JUnit Assertion methods**
  - `junit.framework.TestCase` provides methods for "checking" conditions in a test case.
  - These **assertion** methods are named using the pattern `assert<Condition>(<test>)`, for checking whether the boolean test is true.
  - Common JUnit assertions: (see JUnit test case API)
    - `assertEquals(expr, expr);`
    - `assertFalse(boolean expr);`
    - `assertNotNull(reference);`
    - `assertNull(reference);`
    - `assertTrue(boolean expr);`
Path Testing
- Test cases must be written to ensure that every statement in a class must be executed, (implies that each method must be invoked & tested).

Boundary-Value Testing
- Test cases must be coded to check states that are immediately below, right on and above method execution/data boundaries.
- For example, consider a method that sets the circumference of a person’s head to determine their hat size. Due to the range of hat sizes the company sells the range of acceptable input is limited to MINHEAD…MAXHEAD.
  - Boundary value tests cases must be written to check the result of the method for the following values: MINHEAD-1, MINHEAD, MINHEAD+1, MAXHEAD-1, MAXHEAD, MAXHEAD+1

Equivalence Partitioning
- Test cases are grouped into equivalent partitions & only 1 or 2 cases from within each equivalence partition is tested.
- For the previous example the equivalence partitions would be:
  - values below the minimum, minimum, legal values, maximum, values above the maximum.
Problem

- Code a test class to perform automated unit testing on the Person class.