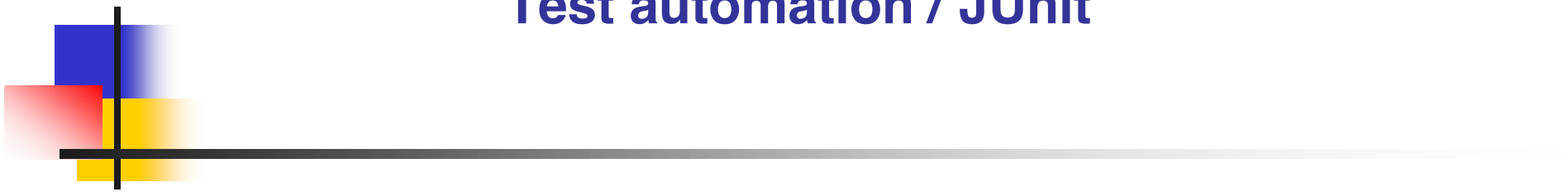
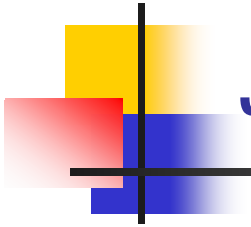


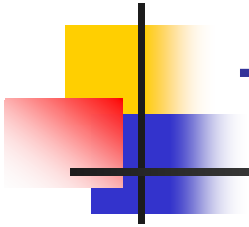
Test automation / JUnit

Building automatically repeatable test suites



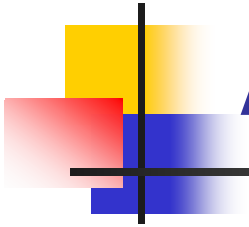
JUnit in Eclipse

- For this course, we will use JUnit in Eclipse
  - **It is automatically a part of Eclipse**
  - **One documentation site (all one line**
    - **www.ibm.com/developerworks/java/tutorials/j-junit4/section5.html**
  - **JUnit can be downloaded from** www.junit.org
- Eclipse contains wizards to help with the development of test suites with JUnit
- JUnit results are presented in an Eclipse window



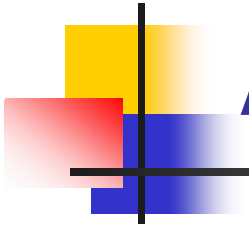
Test automation

- Test automation is software that automates any aspect of testing
  - **Generating test inputs and expected results**
  - **Running test suites without manual intervention**
  - **Evaluating pass/no pass**
- Testing must be automated to be effective and repeatable



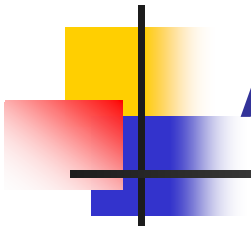
Automated testing steps

- Exercise the implementation with the automated test suite
- Repair faults revealed by failures
- Rerun the test suite on the revised implementation
- Evaluate test suite coverage
- Enhance the test suite to achieve coverage goals
- Rerun the automated test suite to support regression testing



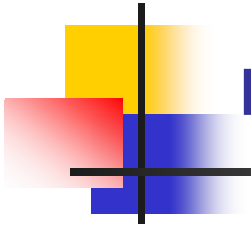
Automated testing advantages

- Permits quick and efficient verification of bug fixes
- Speeds debugging and reduces “bad fixes”
- Allows consistent capture and analysis of test results
- Its cost is recovered through increased productivity and better system quality
- More time to design better tests, rather than entering and reentering tests



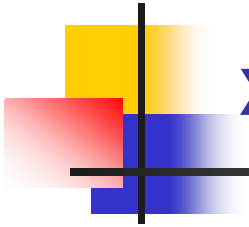
Automated testing advantages

- Unlike manual testing, it is not error-prone and tedious
- Only feasible way to do regression testing
- Necessary to run long and complex tests
- Easily evaluates large quantities of output



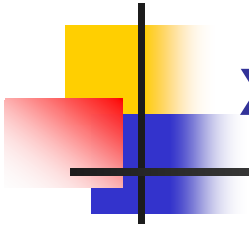
Limitations and caveats

- A skilled tester can use his experience to react to manual testing results by improvising effective tests
- Automated tests are expensive to create and maintain
- If the implementation is changing frequently, maintaining the test suite might be difficult



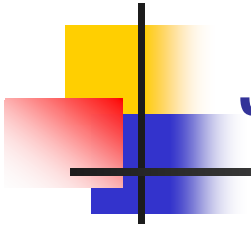
XP approach to testing

- In the Extreme Programming approach
  - **Tests are written before the code itself**
  - **If the code has no automated test cases, it is assumed not to work**
  - **A testing framework is used so that automated testing can be done after every small change to the code**
    - **This may be as often as every 5 or 10 minutes**
  - **If a bug is found after development, a test is created to keep the bug from coming back**



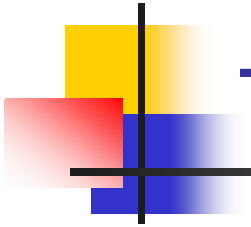
XP consequences

- Fewer bugs
- More maintainable code
- The code can be refactored without fear
- Continuous integration
  - **During development, the program *always works***
  - **It may not do everything required, but what it does, it does right**



JUnit

- JUnit is a framework for writing tests
  - **Written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)**
  - **Uses Java 5 features such as annotations and static imports**
- JUnit helps the programmer:
  - **define and execute tests and test suites**
  - **formalize requirements**
  - **write and debug code**
  - **integrate code and always be ready to release a working version**



Terminology

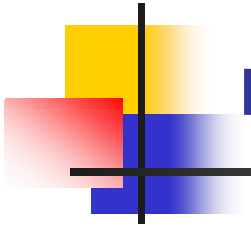
- A **test fixture** sets up the data (both objects and primitives) that are needed for every test
  - **Example: If you are testing code that updates an employee record, you need an employee record to test it on**
- A **unit test** is a test of a *single* class
- A **test case** tests the response of a single method to a particular set of inputs
- A **test suite** is a collection of unit tests
- A **test runner** is software that runs tests and reports results

Example Currency program

```
package currency;
public class Currency {

protected int amount;
protected String type;

Currency(int amt, String typ) { amount = amt;    type = typ;  }

public boolean equals(Object obj) {
  return amount == ((Currency) obj).amount
          && type == ((Currency) obj).type;  }

protected Currency times(int multiplier) {
  return new Currency(amount * multiplier, type);  }

static Currency dollar(int amt) {  return new Currency(amt, "Dollar");  }

static Currency franc(int amt){ return new Currency(amt, "Franc");  }
}
```

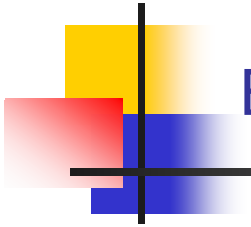
Example Currency test program – 1 of 2

```
package currency;
import org.junit.*;
import static org.junit.Assert.assertTrue;

public class Currency_Test {

@BeforeClass
public static void setUpBeforeClass() throws Exception { }

@AfterClass
public static void tearDownAfterClass() throws Exception { }

@Before
public static void setUp() throws Exception { }

@After
public static void tearDown() throws Exception { }

…
```



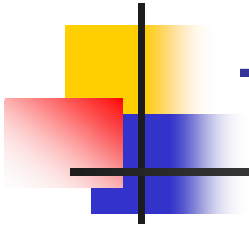
Example Currency test program – 2 of 2

```
...
public void testEquality() {
  assertTrue(new Currency(5, "Franc").equals(new Currency(5, "Franc")));
  assertFalse(new Currency(5, "Franc").equals(new Currency(6, "Franc")));
  assertFalse(new Currency(5, "Franc").equals(new Currency(5, "Currency")));
}
public void testMultiplication() {
  Currency five = new Currency(5, "Dollar");
  assertEquals(new Currency(15, "Dollar"), five.times(3));  }
public void testCurrencyType( )
  assertEquals("Dollar", Currency.dollar(1).type);
  assertEquals("Franc", Currency.franc(1).type);
}
}
```



Example running multiple test classes

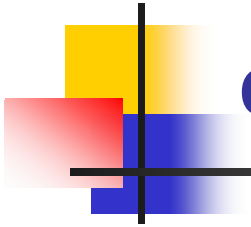
```
package currency;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses( {
  Currency_BoundaryTest.class,
  Currency_EquivalenceTest.class ,
  Currency_DecisionTest.class ,
} )

public class AllTests {
}

}
```



Test fixtures

- Methods annotated with **@Before** will execute before every test case
- Methods annotated with **@After** will execute after every test case
- The routine names are your choice

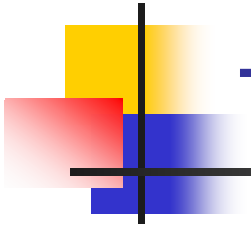
```
@Before
public static void setUp() {…}

@After
public static void tearDown() {…}
```



Class Test fixtures

- Methods annotated with **@BeforeClass** will execute once before all test cases
- Methods annotated with **@AfterClass** will execute once after all test cases
- These are useful if you need to allocate and release expensive resources once

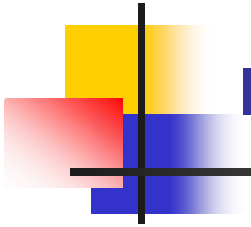
```
@BeforeClass
public static void setUpBeforeClass() {…}

@AfterClass
public static void tearDownAfterClass() {…}
```



Test cases

- Methods annotated with **@Test** are considered to be test cases
  - **Need before every test that you want to execute**

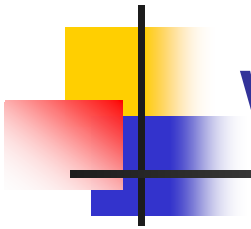
```
@Test
public void test_add() {…}

@Test
public void test_ToString() {…}
```



Ignoring test cases

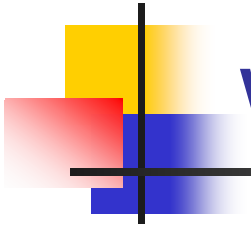
- Test cases that are not to be executed are annotated with `@Ignore`
  - While making corrections due to other test failures
    - Can avoid executing expensive tests
    - Can avoid executing incompletely written tests

```
@Ignore
public void test_add() {…}

@Ignore
public void test_ToString() {…}
```



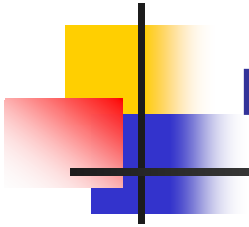
What JUnit does

- For *each* test case **aTestCase**
  - **JUnit executes all @Before methods**
    - **Their order of execution is not specified**
  - **JUnit executes aTestCase**
    - **Any exceptions during its execution are logged**
  - **JUnit executes all @After methods**
    - **Their order of execution is not specified**
- A report for all test cases is presented



Within a test case

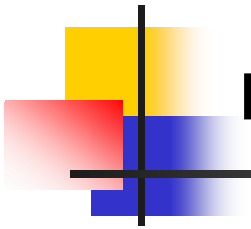
- Call the methods of the class being tested
- Assert what the correct result should be with one of the provided **assert methods**
- These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an **AssertionError** if the test fails
  - **JUnit catches these exceptions and shows you the results**
    - **Only the first failed assert**



List of assert methods 1

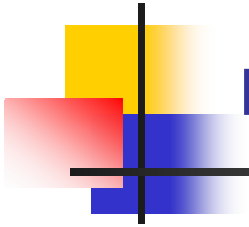
- `assertTrue(boolean b)`
  `assertTrue(String s, boolean b)`
  - Throws an AssertionError if *b* is False
  - The optional message *s* is included in the Error

- `assertFalse(boolean b)`
  `assertFalse(String s, boolean b)`
  - Throws an AssertionError if *b* is True
  - All assert methods have an optional message



Example: Counter class

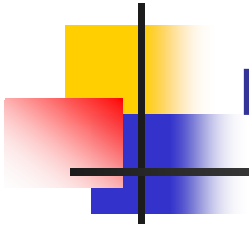
- Consider a trivial “counter” class
  - **The constructor creates a counter and sets it to zero**
  - **The increment method adds one to the counter and returns the new value**
  - **The decrement method subtracts one from the counter and returns the new value**
  - **The corresponding JUnit test class is on the next slide**



Example JUnit test class for counter program

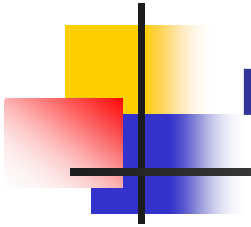
```
public class CounterTest {
    Counter counter1;

    @Before

    public void setUp() {          // create a test fixture
          counter1 = new Counter();
    }

    @Test

    public void testIncrement() {
       assertTrue(counter1.increment() == 1);
       assertTrue(counter1.increment() == 2);
    }

    @Test

    public void testDecrement() {
       assertTrue(counter1.decrement() == -1);
    }

}
```

Each test begins with a *brand new* counter. No need consider the order in which the tests are run.



List of assert methods 2

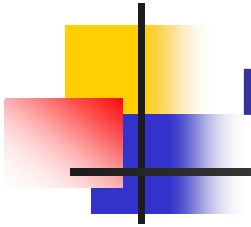
- `assertEquals(Object` ***`expected`***`,`
  `Object` ***`actual`***`)`
  - Uses the equals method to compare the two objects
  - Casting may be required when passing primitives, although autoboxing may be done
  - There is also a version to compare arrays



List of assert methods 3

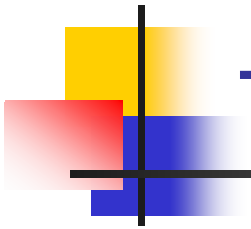
- `assertSame(Object expected, Object actual)`
  - Asserts that two references are attached to the same object (using ==)

- `assertNotSame(Object expected, Object actual)`
  - Asserts that two references are not attached to the same object



List of assert methods 4

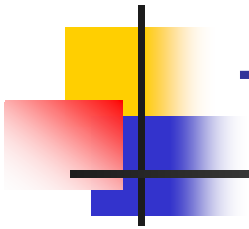
- `assertNull(Object object)`
  - Asserts that a reference is null
- `assertNotNull(Object object)`
  - Asserts that a reference is not null
- `fail()`
  - Causes the test to fail and throw an AssertionError
  - Useful as a result of a complex test, or when testing for exceptions



Testing for exceptions

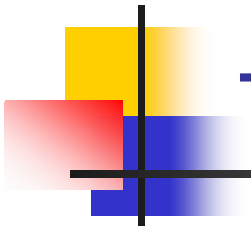
- If a test case is expected to raise an exception, it can be noted as follows and on the next slide

```
@Test(expected = Exception.class)
public void testException() {
   //Code that should raise an exception
   fail("Should raise an exception");
}
```



Testing for exceptions – example

```
public void testAnIOExceptionIsThrown {
   try
   {
      // Code that should raise an IO exception
      fail("Expected an IO exception");
   }
   catch (IOException e)
   {
      // This is the expected result,
      // leave it empty so that the test
      // will pass. If you care about
      // particulars of the exception, you
      // can test various assertions about
      // the exception object
   }
}
```



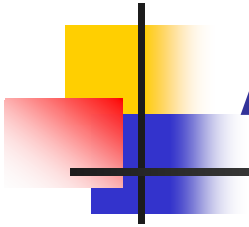
The assert statement

- A statement such as

  assert *boolean_condition*;

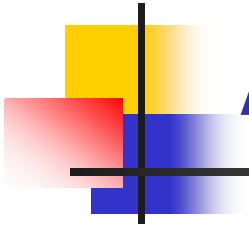
  will also throw an **AssertionError** if the ***boolean_condition*** is false

- Can be used instead of the JUnit **assertTrue** method



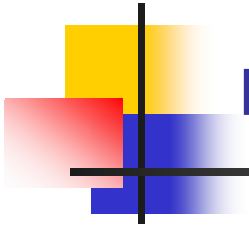
Automated testing issues

- It isn't easy to see how to unit test GUI code
- JUnit is designed to call methods and compare the results they return against expected results
  - **This works great for methods that *just* return results, but many methods have side effects**



Automated testing issues

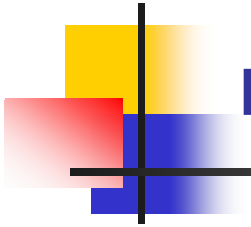
- To test methods that do output, you have to capture the output
  - **It's possible to capture output, but it's an unpleasant coding chore**
- To test methods that change the state of the object, you have to have code that checks the state
  - **It's a good idea to have methods that test state invariants**



First steps toward solutions

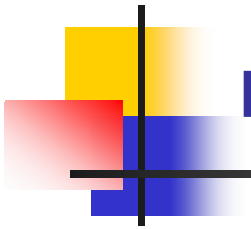
- You can redefine `System.out` to use a different `PrintStream` with
  - `System.setOut(PrintStream)`

- You can “automate” GUI use by “faking” events



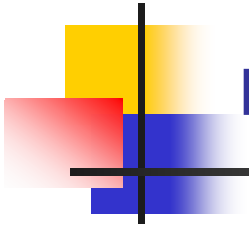
No tool?

- What do you do if there is no equivalent to JUnit for the language or system in which you have to write test cases?



Minimal output testing – 1

- What to do if no tool exists?
  - **Use minimal output testing**
    - **Works for any programming language**
    - **Works for any system**
  - **Successful test outputs only the briefest of messages**
    - **test started**
      test ended



Minimal output testing – 2

- Basic structure
  - **Test program is a sequence of if-statements with the following structure**
    - **Note use of msg_id to identify which test failed**
  - **Rest of test program consists of set up and support routines to simplify programming the condition and the then-phrase**

```
if expected_output ≠ actual output
then print_message(msg_id, … )
fi
```