5 Errors

There are several kind of errors that may occur in programs, generally speaking, and it makes sense to classify them somehow.

(1) Syntax errors — for instance, writing `clas` instead of `class`.

(2) Type errors —

   (a) Applying an operation that works only on some type of values on values of the wrong type. For instance, trying to add two Boolean values, or trying to divide strings.

   (b) Invoking a method `m` on an object that does not define such a method `m`, for instance, if `p` is an instance of `Point`, trying to invoke `p.playTune()` is an error.

(3) Runtime errors —

   (a) Invoking an operation with arguments of the right types for which the operation is undefined. For instance, dividing by 0, or taking the tangent of $\pi/2$.

   (b) Events out of a programmer's control, such as hardware failures. For instance, disk failures during disk IO, network failure during network IO, or the CPU melting. Running out of memory is also something that sometimes occurs.

(4) Logical errors — the program does not behave as expected, such as an implementation of addition that returns 3 when given inputs 1 and 1, or a `distance()` method on points that miscomputes the distance.

These distinctions are not sharply defined; some errors may well be classified in different categories. But the categories are useful as a general sense of the kind of errors that arise.[1]

Different languages check and deal with these errors differently. An error, when it is recognized as such, can either be detected *before* the code is run — in which case we say the error is *detected statically* — or detected *while* the code is run — in which case we say the error is *detected dynamically*.

[1]Some errors do not fit neatly in the above categories. For instance, infinite loops are strange beasts— they often are errors (but not always, since some programs are essentially infinite loops, such as operating systems) but they cannot be detercted during execution, since how do you ever know that your long-running computation is actually an infinite loop as opposed to, say, an actual long but terminating computation?

There are advantages to checking for errors before execution; in particular, you still have a chance to correct the problem while the code is in your hands. With dynamic type checking, errors may only show up after the code has been shipped and the piece of software is in the hands of the customer, making it more difficult and expensive to correct. Static error checking also has some disadvantages. In particular, there is no way to identify exactly all those programs that have errors without actually executing the program.[2] Thus, the error checker needs to approximate, and it will approximate conservatively. Thus, there are programs that would not cause problems during execution that the error checker will reject.

Languages will differ in terms of where they draw the line as to what errors they try to detect statically, and what errors they try to detect dynamically. For example, it is generally the case that the first kind of errors, syntax errors, are detected statically, by a simple parsing pass over the program.

In contrast, runtime errors, errors of type (3), are generally detected during execution, for a variety of reasons: (3a) because they are too difficult to detect statically, and (3b) because it doesn't make sense to try to detect statically errors that occur because something bad happens during execution! Errors of type (4), logical errors, cannot be detected by a language in the first place, either statically or dynamically, since they rely on the notion of "what's expected" which only exists in the designer's or programmer's head. We'll need to do something else for those.

In summary, the main difference between languages, at least with respect to detecting errors, will involve how they deal with errors of type (2), namely type errors.

5.1 Type Errors

When a language checks for type errors dynamically, we say that the language supports *dynamic type checking*. Scheme supports dynamic type checking. Other languages actually can check for type errors at compile time, that is, before programs execute. They support *static type checking*. Java and Scala support static type checking.

In Scala, because everything is in an object, that is, every value is an instance of some class, all type errors amount to errors of type (2b), that is, errors where one is attempting to invoke a method that does not exist.

A type checker is the mechanism that languages such as Scala employ to try to determine if there are type errors in a program. Now, it turns out that it is impossible to exactly recognize, given a specific program P, whether program P will ever attempt to invoke an undefined method during its execution. The reason for this is the same as that hinted at

[2]This is a deep limitation in what we can say about programs in general, which you will see in a good theory of computation course. It is a consequence of the so-called *undecidability of the halting problem for Turing machines*. Very roughly speaking, the limitation is that it is impossible to write a program that takes a program P as input and (without executing the program) answers correctly a question about how P will execute.

in Footnote 2, and the following series examples may help you get a sense for what the difficulties might be.

Consider the following method:

```
def test (x:Boolean, P:Point):Double =
  if (x)
    p.distanceFromOrigin()
  else
    p.explode()
```

What should the type checker do with this method? Should it accept it, or flag it as an error? Note that a `Point` does not have an `explode()` method. This method intuitively causes an error whenever it is called with `false` as a first argument — since then the program will attempt to call method `explode()` on a point, which doesn't exist. If we have the full program, maybe we can figure out whether the method is ever called with a `false` as a first argument. But if we don't — if we're compiling pieces of the program separately, then we cannot determine in advance what the first arguments to `test()` will be, and the type checker has to be conservative and say that because there *might* be an error, it has to reject the program.

But even if we do have the full program in front of us, things are not so easy. Consider the following variant, where instead of taking a Boolean value as first argument, we take an arbitrary integer:

```
def test2 (y:Int, P:Point):Double =
  if (y==0)
    p.distanceFromOrigin()
  else
    p.explode()
```

This causes an error whenever we call `test2()` with a non-zero first argument. Can we always check if the first argument is non-zero? Well, suppose that `test2()` is ever only called once in the program, but in the following way:

```
test2(someComplicatedFunction(10),p)
```

where `p` is a `Point`. This causes an error exactly when `someComplicatedFunction(10)` is non-zero. The point being, `someComplicatedFunction()` may be arbitrarily complicated, involving for instance some non-trivial number theory, and it is completely unreasonable to expect a type checker to analyze `someComplicatedFunction()` carefully and figure out that when given 10, it returns a non-zero number. So here, again, the type checker will have to be conservative and say that because `test2` *may* cause an error (because function `someComplicatedFunction()` is too complicated to analyze), it will report a type error.

So the type checker is really an approximation to detecting type errors that may occur during execution. This approximxation is conservative. It will reject programs if there is a reason to believe that they *may* cause type errors. This is made precise by the following property of the Scala type systems, called *type safety*.

Type Safety; If a Scala program P type checks (that is, passes the type checker), then no execution of P ever attempts to invoke a method m on a (non-null) object that does not define m.

We say a program P is safe if no execution of P ever attempts to invoke a method m on a non-null object that does not define m. Thus, type safety says that if a Scala program P type checks, then P is safe.

Note the direction of the implication. It may well be the case that P never attempts to invoke a method m on an object that does not define m, but that type checker does not recognize that situation and rejects the program. That's not uncommon. In fact, the above says less than one might hope. In particular, a really dumb type checker that simply rejects all programs satisfies type safety. (Why?) But thankfully, we know that the Scala type checker does accept some programs, so it is not completely vacuous.

It is important to keep in mind the property above, that the type system guarantees safety. Because that's the only thing that the type system enforces. And the name of the game will be trying to get the most programs to type check, and still be guaranteed safety. So when we look at the ways Scala has of reusing code, we will make sure to check in what way those techniques still ensure safety. And when Scala forbids us from doing something, we will be able to check in what way that something could be used to subvert safety.

5.2 Runtime Errors

Errors of type (2) are errors that are generally detected during execution. Every language will have a different way to report those errors. Generally, the error will abort execution and report a useful error message on the console. But in many languages, Scala included, these errors can be dealt with within the program itself, and the execution need not actually abort. In other words, these errors can often be recovered from gracefully.

In medevial times, one way to report errors at run time was to have functions return a special value called various an *error value*, or an *error flag*. Basically, a special value that did not make sense as a return value for the function, and that could be used to indicate that execution failed. For instance, a function could return -1 to indicate an error instead of the usual positive number it would return on a correct execution.

The problem with error values is that it is very easy to forget to check the result of a function call to see if an error occurred. And forgetting to check the result of a function call means that if there was indeed an error, the error value gets propagated in your code and will

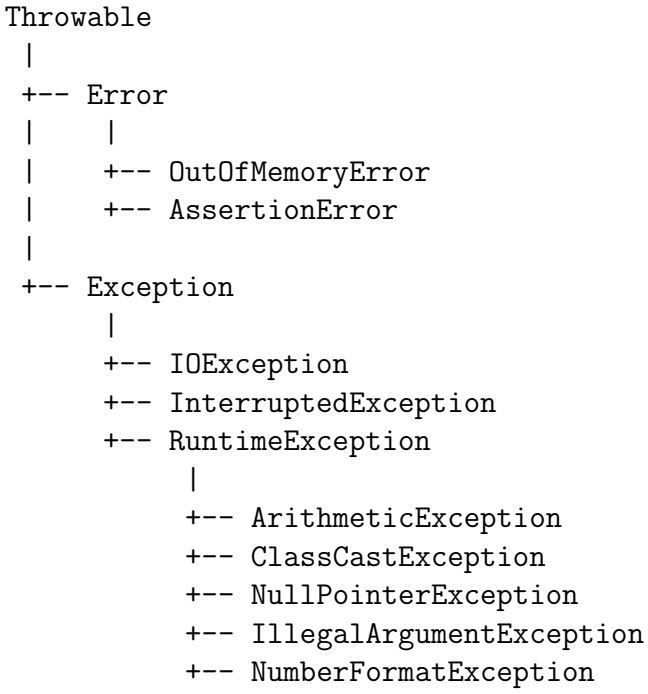

Figure 1: Exceptions Class Hierarchy

likely cause problems somewhere else (e.g., if you end up taking the square root of the value returned by the function, and that value was the error value -1). Tracking these kinds of bugs is difficult.

This error-value technique is still in use nowadays, unfortunately. In Scala, for instance, the *null* value is often used as an error marker — the *null* value is a special object that can be considered an instance of (essentially) every class. The problem is that, as I mentioned, it is very easy to forget to check that a value is not *null* when returned from a function, and that *null* value can be propagated in the code, and cause problems when someone tries to invoke a method on *null*, which does not implement any method. I strongly suggest you avoid using *null* for error reporting — in fact you should probably avoid *null* altogether. (We'll see techniques for doing so later.)

The error-value technique has the consequence that if you don't check that an error occurred, that error is free to wreak havoc in the rest of your program. That's bad, because programmers are lazy and will forget to check for errors.

The opposite point of view is taken in the modern approach to dealing with run-time errors, *exceptions*. An exception is just an object in the system, that gets created and *thrown* when an error is encountered. This immediately terminates the currently executing method. The methods caller can either *catch* and *handle* that exception, or it can itself simply terminate, in which case the exception propagates to the callers caller. The exception propagates in

this way, unwinding the call stack, until a method handles it or there are no more methods left. Thus, if an exception is not caught, instead of letting your code continue and cause further problem, your whole program aborts. If you want your code to keep going — perhaps because you have another way to try to achieve your goal, such as trying to write on the network if writing to disk failed — then you can catch the exception and deal with it.

Let's look at how Scala implements exceptions. There is actually a whole hierarchy of classes in Scala implementing exceptions. This hierarchy lets us distinguish the kind of exceptions that can occur. Figure 1 presents the (partial) class hierarchy of exceptions.

Class `Throwable` is the most general kind of exception that every other exception subclasses. The `Error` class roughly represent the show-stoppers, that lead to aborting execution in almost all cases. The `Exception` class capture more "benign" forms of errors. These include `IOException`s, representing exceptions due to failure of IO (disk failure, network failure, and so on), while `RuntimeException`s represent exceptions such as dividing by 0 (an `ArithmeticException`), casting an object to an unacceptable class (a `ClassCastException`), invoking a method on a null object (a `NullPointerException`). The `IllegalArgumentException` is a general exception to represent passing a wrong value to a method.

(Note that you can also create new kinds of exceptions by subclassing, generally by subclassing the `Exception` class. We'll see subclassing more extensively in the coming weeks.)

Many exceptions are created automatically by the system when an error is encountered — for instance, if you try to invoke a method on a *null* value. You can also throw an exception yourself in the code, as in:

```
throw new IllegalArgumentException("Oops, something bad happened");
```

This is all good and well, but how can we deal with such exceptions gracefully? After all, if exceptions were just errors that cannot be dealt with, we could just say that an exception aborts execution, and be done with it. The idea is that if a method call appear in the body of a `try` block, and if that call returns an exception, that exception is not returned immediately, but instead it is handled `catch` clause associated with the `try` block. This `catch` clause uses pattern matching (of the kind we saw in the `equals()` method implementation last lecture) to do different things depending on the kind of execution that was caught.

Suppose `MyException` is a new exception you have defined and suppose you wanted to intercept this exception if it is thrown by the method `someMethod()` on object `obj`:

```
try {
  obj.someMethod();
} catch {
  case ex : MyException => // some code to deal with the exception
}
```

This reads: if a `MyException` is thrown during the execution of `obj.someMethod()`, then instead of having that exception propagate up, intercept it, name it `ex`, and continue execution of the current method with the code in the corresponding case of the `catch` clause. (The reason why we may be interested in `ex` is that `ex` is an object that actually implements some useful methods such as `getMessage()` which returns a string representing the message associated with the exception.) This lets you gracefull recover from an exception, by intercepting it and dealing with it instead of letting it bubble up until it aborts the entire program with an error.

Note that a `catch` clause catching an exception `SomeException` will in fact catch all exceptions that are instances of subclasses of `SomeException`. This lets you intercept, for instance, any possible exception in a single catch clause:

```
try {
  // some code doing something interesting
} catch {
  case ex : Throwable => // deal with the exception
}
```

This works because every exception ultimately subclasses `Throwable`. This kind of code is useful in testing code to nicely format the exception and report useful information.

It is also possible to catch multiple exceptions and dealing with them differently:

```
try {
  // some code doing something interesting
} catch {
  case ex : SomeExceptionClass =>
               // deal with this kind of exception
  case ex : SomeOtherExceptionClass =>
               // deal with this other kind of exception
}
```

The order of the cases in the `catch` clauses is relevant — they are tried in order, and the first case where the current exception is in a subclass of the specified exception will be selected.

5.3 Logical Errors

Errors of type (4), logical errors — programs not behaving as expected — are the most difficult to catch and correct, if only because languages do not offer any facilities for identifying those errors. They rely on the notion of "expected program behavior", which is really only in the head of the designer or the programmer.

As far as ADTs are concerned, we know what behavior to expect: it's exactly the behavior described by the specification. So how do we check that a program satisfies its specification (i.e., behaves as expected)? There are two main ways: formal verification, and software testing.

5.3.1 Testing versus Formal Verification

Formal verification, what you learned in *Logic and Computation*, is the process of verifying using logic or other means that a program satisfies its specification. As you will remember from *Logic and Computation*, formal verification is usually difficult. The purpose of formal verification is proving a program correct (i.e., satisfies its specification) for all possible inputs. The point of testing, on the other hand, is to *find bugs*, and not to prove programs correct. Why?

To prove that a program is correct using tests, we need to exercise all the inputs to the program. This can be a problem if there many inputs to test. For instance, a program that takes two 64-bit integers as input, that is, 128 bits of input total, would require 10^{22} years to run through tests for all inputs, even if we allow a millon tests to be performed every second. That's clearly infeasible. Things get worse, of course, because some programs can take one of infinitely many inputs — for instance, programs manipulating lists.

So we cannot in general exercise all inputs, and therefore need to focus only on a few. This leaves the possibility of bugs for the other inputs.

Consider the well-publicized story of the Intel FDIV bug. In 1994, Intel recalled its Pentium processors to repair a bug in its floating point division instruction, FDIV. Intel has estimated that this recall cost the company 475 million dollars.

Here is an example of the bug:

> 4195835.0/3145727.0 = 1.333 820 449 136 241 000 (Correct value)
>
> 4195835.0/3145727.0 = 1.333 739 068 902 037 589 (Flawed Pentium)

Intel estimated that the result of the FDIV instruction was incorrect a little more often than once in every 9 billion floating point divisions. An IBM study found, however, that the values that occur most often in spreadsheet and scientific computations are more likely to trigger the bug.

After the Intel debacle, AMD, Intel's chief competitor, was worried about their own chip. They turned to the automatic theorem proving community, and Moore, a leading researcher in the area, proved with his team that the algorithm used to implement the FDIV instruction in AMD's chip was correct using a mechanical theorem prover. They in fact managed to prove the correctness of the entire floating-point kernel (not just the FDIV instruction). In the process, they found and repaired two design errors that had not been caught by any of the 80 million separate tests that had been run.

Unfortunately, as I said, proving programs correct is quite hard, so testing is often advocated as a first approach to debugging. We may return to proving programs correct towards the end of the course, but for the time being, we focus on testing.

5.3.2 White-Box (or Glass-Box) versus Black-Box Testing

Testing techniques can be classified along several orthogonal dimensions.

White-box testing relies on knowing the internals of the implementation. For instance, testing by inserting print statements or something similar to get a sense of what gets executed is a form of white-box testing. This is useful when you are debugging a particular implementation of an ADT, but it of course specific to that particular implementation.

A particularly useful technique for white-box testing is to use assertions. Let's take assertions in Scala as an example. An *assertion* takes the following form:

`assert`(*BooleanExpression*)

When Scala encounters an `assert` during execution, it will evaluate the *BooleanExpression* in the assertion; if it evaluates to true, then execution proceeds to whatever statement follows the assertion; if it evaluates to false, then an exception `AssertError` is thrown.

In short, assertions provide a way to put in "sanity checks" within your own code. Generally speaking, assertions are meant to check invariants that *must* be true in order for the code to work correctly. This means, in particular, that whenever possible, you should have assertions on when you ship your code out. (There are ways to make sure that Scala or whatever language you use executes your code with assertions turned on. Please refer to the documentation.) Many programmers do not enable assertions on production code, for fear generally of the run-time cost of checking all those assertions. Unless your assertions are very heavy, however, I would recomment you keep assertions on. The benefit of having this extra layer of checks for the important invariants your code usually outweighs the minor inconvenience of a slightly slower execution. (And, let's face it, if you are programming in Scala or Java, you are already more or less admitting that executing programs as fast as possible is not your priority.)

In contrast to white-box testing, black-box testing treats the implementation as hidden, and can only test the code based on its interface and its specification. (It treats the ADT as a black box — you do not get to peek inside.) The advantage of black-box testing is that it can be used to test *any* implementation of the ADT.

The testing you performed in *Fundies I* and *II*, using test cases, is a form of black-box testing.

A black-box testing infrastructure for an ADT *must be able to test any implementation of the ADT, and any correct implementation must pass the test.*

5.3.3 Unit Versus Integration Testing

A different axis of comparison for testing approaches, orthogonal to the white-box/black-box axis, is to consider whether we are testing pieces of the program in isolation or as a whole. Let me try to make that a bit more precise.

A class or a closely related set of classes (module) is often implemented by a single programmer. The class or module all by itself is not runnable, and generally will not do anything useful on its own. It is used within the context of a larger piece of software. A key feature of the ADT approach we advocate in this course is that it allows the class to be implemented without regard to the context in which it will be used in.

(This is in fact the whole point of specification-based design in the wider context of engineering. By designing a piece according to a specification, other pieces needing to interact with the piece need only assume that the piece satisfies its specification. The actual details of how the specification is met — the implementation — does not matter.)

Unit testing is the process of taking a class implementation and making sure it satisfies its specification. This requires making up test data and a test program to test the implementation against the specification.

In contrast, integration testing tests the class in a context with other classes to make sure that the higher-level interaction between the classes works correctly.

In this course, because of our ADT design philosophy, we will concentrate on black-box unit testing.

Yet another sort of testing that is sometimes mentioned is *regression testing*. Roughly speaking, regression testing is the process of keeping old tests around; because adding a new feature to a piece of software should not change existing behaviors, we can run the old tests to ensure that the code still behaves as expected.

5.3.4 Test Coverage

Given that we can only test a finite (and in fact small) number of cases, how do we choose those tests?

A good test is a test that finds bugs. Finding good tests therefore requires a basic understanding of the common bugs that can occur. This depends on the actual programming language used, and on the kind of program being tested. Many books have been written on common bugs in various programming languages.

In general, test coverage should include:

- Trivial cases (e.g., empty list inputs for list-processing functions)
- Typical cases (both easy and hard)

- Boundary cases (cases that are either just acceptable, or just outside acceptable; e.g., inputs that access an array close to its bounds)[3]
- Weird cases
- Error cases (if the specification mentions how error cases are treated)

What should be tested during a test? Every equation in the specification should be tested. You may also want to check the implicit specifications of some of the methods that are Scala-specific, such as `equals()`, which is required to be reflexive, symmetric, and transitive; or `hashCode()`, which is required to satisfy that `a.equals(b)` implies `a.hashCode()==b.hashCode()`.

5.3.5 Example: Black-Box Unit Testing Infrastructure for ADTs

Let us look at an example of a black-box unit testing class for the POINT ADT we've been playing with. Recall the signature:

```
CREATORS
  cartesian : (Double, Double) -> Point
  polar :     (Double, Double) -> Point

OPERATIONS
  xCoord :              () -> Double
  yCoord :              () -> Double
  angleWithXAxis :      () -> Double
  distanceFromOrigin :  () -> Double
  distance :            (Point) -> Double
  move :                (Double, Double) -> Point
  add :                 (Point) -> Point
  rotate :              (Double) -> Point

  isEqual :             (Point) -> Boolean
  isOrigin :            () -> Boolean
```

and the specification:

$$\texttt{cartesian}(x,y)\texttt{.xCoord()} = x$$
$$\texttt{polar}(r,\theta)\texttt{.xCoord()} = r\cos\theta$$

$$\texttt{cartesian}(x,y)\texttt{.yCoord()} = y$$

[3] In some cases, boundary cases may depend on the specifics of an implementation of an ADT; this sort of boundary case has a flavor of white-box testing.

$\texttt{polar}(r,\theta)\texttt{.yCoord()} = r\sin\theta$

$\texttt{cartesian}(x,y)\texttt{.distanceFromOrigin()} = \sqrt{x^2+y^2}$
$\texttt{polar}(r,\theta)\texttt{.distanceFromOrigin()} = r$

$$\texttt{cartesian}(x,y)\texttt{.angleWithXAxis()} = \begin{cases} \tan^{-1}(y/x) & \text{if } x \neq 0 \\ \pi/2 & \text{if } y \geq 0 \text{ and } x = 0 \\ -\pi/2 & \text{if } y < 0 \text{ and } x = 0 \end{cases}$$

$\texttt{polar}(r,\theta)\texttt{.angleWithXAxis()} = \theta$

$\texttt{cartesian}(x,y)\texttt{.distance}(q) = \sqrt{(x - q\texttt{.xCoord()})^2 + (y - q\texttt{.yCoord()})^2}$
$\texttt{polar}(r,\theta)\texttt{.distance}(q) =$
$$\sqrt{(p\texttt{.xCoord()} - q\texttt{.xCoord()})^2 + (p\texttt{.yCoord()} - q\texttt{.yCoord()})^2}$$

$\texttt{cartesian}(x,y)\texttt{.move}(dx,dy) = \texttt{cartesian}(x+dx, y+dy)$
$\texttt{polar}(r,\theta)\texttt{.move}(dx,dy) = \texttt{cartesian}(r\cos\theta + dx, r\sin\theta + dy)$

$\texttt{cartesian}(x,y)\texttt{.add}(q) = \texttt{cartesian}(x + q\texttt{.xCoord()}, y + q\texttt{.yCoord()})$
$\texttt{polar}(r,\theta)\texttt{.add}(q) = \texttt{cartesian}(r\cos\theta + q\texttt{.xCoord()}, r\sin\theta + q\texttt{.yCoord()})$

$\texttt{cartesian}(x,y)\texttt{.rotate}(\rho) = \texttt{cartesian}(x\cos\rho - y\sin\rho, x\sin\rho + y\cos\rho)$
$\texttt{polar}(r,\theta)\texttt{.rotate}(\rho) = \texttt{polar}(r, \theta+\rho)$

$$\texttt{cartesian}(x,y)\texttt{.isEqual}(q) = \begin{cases} \mathit{true} & \text{if } x = q\texttt{.xCoord()} \text{ and } y = q\texttt{.yCoord()} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isEqual}(q) = \begin{cases} \mathit{true} & \text{if } r = q\texttt{.distanceFromOrigin()} \text{ and} \\ & \quad \theta \equiv q\texttt{.angleWithXAxis()} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{cartesian}(x,y)\texttt{.isOrigin()} = \begin{cases} \mathit{true} & \text{if } x = 0 \text{ and } y = 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isOrigin()} = \begin{cases} \mathit{true} & \text{if } r = 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

(Where $\equiv$ for angles is defined in Lecture 2.)

Assume we have a module `Point` containing the creators of the ADT.

The general structure of the tester is to first generate a set of instances of `Point` for each of the creators, and test each instance against the specification. The instances, at least for the typical cases, can be usefully generated at random.

First off, the module definition and `main` method, which invokes the `test_all()` method that runs the battery of tests.

```
object PointTester {

  // The main entry point of the tester

  def main (argv:Array[String]):Unit = test_all()

  def test_all ():Unit = {
    var failed = 0;

    failed += test_cartesian()
    failed += test_polar()

    println("\nNumber of tests failed: " + failed)
  }
```

The `test_all()` method runs the tests for each of the creators: `test_cartesian()` and `test_polar()`, each in charge of generating random points using the respective creators and running the appropriate tests.

In order to creates point at random, we need helper functions for randomly generating data — you can look-up the details of the class `scala.util.Random` in the documentation for the Scala Standard Library.

```
  private val rnd = new scala.util.Random

  private def randomDouble ():Double =
    1000 * rnd.nextDouble()

  private def randomAngle ():Double =
    2 * math.Pi * rnd.nextDouble()

  private def randomPoint ():Point =
    if (rnd.nextBoolean())
      Point.cartesian(randomDouble()-500,randomDouble()-500)
    else
      Point.polar(randomDouble(),randomAngle())
```

We can now define `test_cartesian()` and `test_polar()` that randomly generate a given number of points using the corresponding creators, and call the various testing functions for those generated points. Note that exceptions are caught and reported as failed tests.

```
// These are the tests for the creators
// For each creator, we call the tests for operations on that creator
// The result is the number of tests that failed

val numberRandomPoints = 100

private def test_cartesian ():Int = {
  var failed : Int = 0
  for (i <- 1 to numberRandomPoints) {
    val x = randomDouble()-500
    val y = randomDouble()-500
    try {
      failed += test_xCoord_cartesian(x,y)
      failed += test_yCoord_cartesian(x,y)
      failed += test_distanceFromOrigin_cartesian(x,y)
      failed += test_angleWithXAxis_cartesian(x,y)
      failed += test_distance_cartesian(x,y)
      failed += test_move_cartesian(x,y)
      failed += test_add_cartesian(x,y)
      failed += test_add_cartesian(x,y)
      failed += test_rotate_cartesian(x,y)
      failed += test_isEqual_cartesian(x,y)
      failed += test_isOrigin_cartesian(x,y)
    } catch {
      case ex: RuntimeException => {
        // If there was an exception anywhere in there, then we
        //   have a problem
        assertT(false, "Exception: "+ex.getMessage(),"","")
        return failed+1;
      }
    }
  }
  return failed
}

private def test_polar ():Int = {
  var failed : Int = 0
  for (i <- 1 to numberRandomPoints) {
    val r = randomDouble()
```

```
      val t = randomAngle()
      try {
        failed += test_xCoord_polar(r,t)
        failed += test_yCoord_polar(r,t)
        failed += test_distanceFromOrigin_polar(r,t)
        failed += test_angleWithXAxis_polar(r,t)
        failed += test_distance_polar(r,t)
        failed += test_move_polar(r,t)
        failed += test_add_polar(r,t)
        failed += test_add_polar(r,t)
        failed += test_rotate_polar(r,t)
        failed += test_isEqual_polar(r,t)
        failed += test_isOrigin_polar(r,t)
      } catch {
        case ex: RuntimeException => {
          // If there was an exception anywhere in there, then we
          //   have a problem
          assertT(false, "Exception: "+ex.getMessage(),"","")
          return failed+1;
        }
      }
    }
    return failed
  }
```

To actually perform the tests, we will need helper functions to perform various comparisons — for instance, we will want to check that two `Double` are close enough to each other (to account for possible round-off errors), as well as checking for two angles being equal by taking into account the fact that angle values wrap at every 2π intervals. Finally, we will also want to compare points for equality without relying on the implementation of equality as defined in the ADT itself, since we are testing that particular implementation.

```
  //  Our own definition of equality
  //  Still depends on the provided isX() methods from implementation

  private val tolerance : Double = 0.00001

  private def almost (d1 : Double, d2 : Double):Boolean =
    (d1-d2 < tolerance) && (d1-d2 > -tolerance)

  private def eqPoints (p : Point, q : Point):Boolean =
    (almost(p.xCoord(),q.xCoord()) && almost(p.yCoord(),q.yCoord()))
```

```
private def normalize (angle:Double):Double =
  if (angle >= 2*math.Pi)
    normalize(angle-2*math.Pi)
  else if (angle < 0)
    normalize(angle+2*math.Pi)
  else
    angle

private def eqAngles (a : Double, b : Double):Boolean =
  almost(normalize(a),normalize(b))
```

We can now define our tests. For every equation in the specification (which corresponds to a given operation *op* applied to the resulting value of a given creator *cr*), we have a method `test_`*po*`_`*cr*`()` that checks that the corresponding equation holds. Note that these methods take as arguments the arguments to be passed to the creator — the method is in charge of creating an instance of a point using those values (and the corresponding creator) and perform the test.

```
// These are individual tests, testing each equation in the
 specification
// The format of the test name is 'test_Operation_Creator'
// Each test takes values needed to create an instance of the ADT
//   via the requested creator
// Each test returns 0 if the test succeeds and 1 if the test fails

private def test_xCoord_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val got = p.xCoord()
  val exp = x
  assertT(almost(got,exp),
          "cartesian("+x+","+y+").xCoord()",
          exp,got)
}

private def test_yCoord_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val got = p.yCoord()
  val exp = y
  assertT(almost(got,exp),
          "cartesian("+x+","+y+").yCoord()",
          exp,got)
}
```

```
private def test_distanceFromOrigin_cartesian (x:Double, y:Double):Int
 = {
  val p = Point.cartesian(x,y)
  val got = p.distanceFromOrigin()
  val exp = math.sqrt(x*x+y*y)
  assertT(almost(got,exp),
          "cartesian("+x+","+y+").distanceFromOrigin()",
          exp,got)
}

private def test_angleWithXAxis_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val got = p.angleWithXAxis()
  val exp = math.atan2(y,x)
  assertT(eqAngles(got,exp),
          "cartesian("+x+","+y+").angleWithXAxis()",
          exp,got)
}

private def test_distance_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val q = randomPoint()
  val got = p.distance(q)
  val exp = math.sqrt(math.pow(x-q.xCoord(),2)+
                      math.pow(y-q.yCoord(),2))
  assertT(almost(got,exp),
          "cartesian("+x+","+y+").distance("+q+")",
          exp,got)
}


private def test_move_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val dx = randomDouble()-500
  val dy = randomDouble()-500
  val got = p.move(dx,dy)
  val exp = Point.cartesian(x+dx,y+dy)
  assertT(eqPoints(got,exp),
          "cartesian("+x+","+y+").move("+dx+","+dy+")",
          exp,got)
}
```

```
private def test_add_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val q = randomPoint()
  val got = p.add(q)
  val exp = Point.cartesian(x+q.xCoord(),y+q.yCoord())
  assertT(eqPoints(got,exp),
          "cartesian("+x+","+y+").add("+q+")",
          exp,got)
}

private def test_rotate_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val t = randomAngle()
  val got = p.rotate(t)
  val exp = Point.cartesian(x*math.cos(t)-y*math.sin(t),
                            x*math.sin(t)+y*math.cos(t))
  assertT(eqPoints(got,exp),
          "cartesian("+x+","+y+").rotate("+t+")",
          exp,got)
}

private def test_isEqual_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val q = randomPoint()
  val got = p.isEqual(q)
  val exp = eqPoints(p,q)
  assertT(got==exp,
          "cartesian("+x+","+y+").isEqual("+q+")",
          exp,got)
}

private def test_isOrigin_cartesian (x:Double, y:Double):Int = {
  val p = Point.cartesian(x,y)
  val got = p.isOrigin()
  val exp = (almost(x,0) && almost(y,0))
  assertT(got==exp,
          "cartesian("+x+","+y+").isOrigin()",
          exp,got)
}
```

```
private def test_xCoord_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val got = p.xCoord()
  val exp = r * math.cos(theta)
  assertT(almost(got,exp),
          "polar("+r+","+theta+").xCoord()",
          exp,got)
}

private def test_yCoord_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val got = p.yCoord()
  val exp = r * math.sin(theta)
  assertT(almost(got,exp),
          "polar("+r+","+theta+").yCoord()",
          exp,got)
}

private def test_distanceFromOrigin_polar (r:Double, theta:Double):Int
 = {
  val p = Point.polar(r,theta)
  val got = p.distanceFromOrigin()
  val exp = r
  assertT(almost(got,exp),
          "polar("+r+","+theta+").distanceFromOrigin()",
          exp,got)
}

private def test_angleWithXAxis_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val got = p.angleWithXAxis()
  val exp = theta
  assertT(eqAngles(got,exp),
          "polar("+r+","+theta+").angleWithXAxis()",
          exp,got)
}

private def test_distance_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val q = randomPoint()
  val got = p.distance(q)
  val exp = math.sqrt(math.pow(p.xCoord()-q.xCoord(),2)+
```

```
                      math.pow(p.yCoord()-q.yCoord(),2))
  assertT(almost(got,exp),
          "polar("+r+","+theta+").distance("+q+")",
          exp,got)
}


private def test_move_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val dx = randomDouble()-500
  val dy = randomDouble()-500
  val got = p.move(dx,dy)
  val exp = Point.cartesian(p.xCoord()+dx,p.yCoord()+dy)
  assertT(eqPoints(got,exp),
          "polar("+r+","+theta+").move("+dx+","+dy+")",
          exp,got)
}

private def test_add_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val q = randomPoint()
  val got = p.add(q)
  val exp = Point.cartesian(p.xCoord()+q.xCoord(),
                            p.yCoord()+q.yCoord())
  assertT(eqPoints(got,exp),
          "polar("+r+","+theta+").add("+q+")",
          exp,got)
}

private def test_rotate_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val t = randomAngle()
  val got = p.rotate(t)
  val exp = Point.polar(r,theta+t)
  assertT(eqPoints(got,exp),
          "polar("+r+","+theta+").rotate("+t+")",
          exp,got)
}

private def test_isEqual_polar (r:Double, theta:Double):Int = {
  val p = Point.polar(r,theta)
  val q = randomPoint()
```

```
    val got = p.isEqual(q)
    val exp = eqPoints(p,q)
    assertT(got==exp,
            "polar("+r+","+theta+").isEqual("+q+")",
            exp,got)
  }

  private def test_isOrigin_polar (r:Double, theta:Double):Int = {
    val p = Point.polar(r,theta)
    val got = p.isOrigin()
    val exp = (almost(r,0))
    assertT(got==exp,
            "polar("+r+","+theta+").isOrigin()",
            exp,got)
  }
```

Finally, the testing code relies on a method `assertT()` that report whether a test succeeded or failed, and in the latter case what were the expected values and the values obtained. This method is a generic method, which we'll cover in more detail later in the course.

```
    /*   Result is expected to be true for passing tests, and false for
     *   failing tests.  If a test fails, we print out the provided
     *   message so the user can see what might have gone wrong.
     *
     * We return 0 if the test succeeded, and 1 if it failed
     *
     * Be sure to review anything that doesn't make sense. */

    private def assertT[A] (result:Boolean, msg:String, exp:A, got:A):Int
    = {
        if (!result) {
          println("\n**TEST FAILED** "+ msg)
          println("   Got: "+got)
          println("   Expected: "+exp)
        }
        else
          print(".")
        return (if (result) 0 else 1)
    }
}
```