Static and Dynamic Errors

Consider the kind of errors that can occur in programs. Let me abstract from Java for the moment, and speak in general.

There are various errors possible, and it makes sense to classify them somehow.

(1) Type errors, that is, trying to apply an operation that works only on one type to values of the wrong type. For instance, trying add Boolean values, or divide strings. In Java, we also include in this class trying to invoke a method on an object that does not define it.[1]

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

(3) Other undefined behavior for the language at hand. For instance, in Java, invoking a method on a null object, or casting an object to a class that is not a superclass of the initial class of the object.

(4) Problems out of a programmer's control, such as hardware failures. For instance, disk failures during disk IO, or network failure during network IO.

(Note that 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.)

Different languages check and deal with these errors differently. Let's focus on the first, type errors. Languages differ in whether they check for type errors at compile time, that is, before programs execute, or at run time, that is, during program execution. In the former case, we say the language supports *static type checking*, while in the latter case, it supports *dynamic type checking*. Java supports static type checking. Scheme, on the other hand, supports dynamic type checking.

There are advantages to checking for type errors at compile time; 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

[1] We view this as a type error because a class is a type, and specifies which methods can be called on values of that type (i.e., objects).

is in the hands of the customer, making it more difficult and expensive to correct. Static type checking also has some disadvantages. In particular, there is no way to identify exactly all those programs that have type errors at compile time.[2] Thus, the type checker needs to approximate, and it will approximate conservatively. Thus, there are programs that are in fact type correct (that would not cause a problem during execution) but that the type checker will report as having a type error.

So, Java takes care of type errors at compile time. The other kind of errors, however, cannot be reliably checked for at compile time. (Here again, the limitation in the footnote above bites us here.) Every language will have a different way to report those kind of errors. Generally, the error will abort execution and report a useful error message on the console. But in many languages, Java 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.

Exceptions

Java uses *exceptions* to report errors and deal with them. An exception is just an object in the system, that gets created when an error is encountered, and that propagates through the code until either it is handled, or aborts execution.

There is actually a whole hierarchy of classes implementing exceptions. This hierarchy lets us distinguish the kind of exceptions that can occur. Here is a partial class hierarchy of exceptions:

```
Throwable
 |
 +-- Error
 |    |
 |    +-- OutOfMemoryError
 |    +-- Assertion Error
 |
 +-- Exception
      |
      +-- IOException
      +-- Interrupted Exception
      +-- RuntimeException
           |
           +-- ArithmeticException
```

[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. 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. Here, the program that checks a property is the type checker.

```
+-- ClassCastException
+-- NullPointerException
+-- IllegalArgumentException
+-- NumberFormatException
```

The 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—this is the exception you used in the first homework to report the error of creating a classroom with a negative number of seats.

New kinds of exceptions can be created by subclassing; generally, one subclasses the `Exception` class.

Many exceptions are created automatically by the system when an error is encountered. You can also create an exception yourself in the code. This is called throwing an exception:

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

Intuitively, when an exception is thrown in a method, either automatically by the system or by calling `throw` directly, this exception will "bubble up" the stack of calling methods. What I mean is that the current method will stop execution, and report to its caller that an exception was thrown. The caller method, by default, will also abort execution, and report to its own caller that an exception was thrown, and so on, until the exception bubbles up to the `main` method that started the program, which itself gets aborted and the exception is reported to the user.

Java distinguishes between checked and unchecked exceptions. Unchecked exceptions (including `Error`s and `RuntimeException`s) are exceptions that can occur essentially at any point during execution. Checked exceptions (including `IOException`s) and all exceptions that you will create) are exceptions that can occur only when specific methods throw them. You are required to annotate every method that can throw a checked exception, either because it can throw it directly, or because it invokes a method that can throw that exception.

```
public void someMethod () throws SomeException {
  // some code that can throw the SomeException exception
}
```

This is all good and well, but how can we handle such exceptions gracefully? Well, as I said, a thrown exception will bubble up the stack of calling methods. Except, if any of the

methods in the stack is wrapped in a `try` block, then the exception can be handled by the `catch` clause associated with the `try` block.

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

```
try {
  obj.someMethod();
} catch (MyException e) {
  // some code to deal with the exception
}
```

This says: if a `MyException` is throw while you are executing `obj.someMethod()`, then instead of having that exception abort the current method call and bubble up, intercept it, call it `e`, and continue execution of the current method with the code in the `catch` clause. (The reason why we may be interested in `e` is that `e` 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 (Throwable e) {
  // 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 (SomeExceptionClass e) {
  // deal with this kind of exception
} catch (SomeOtherExceptionClass e) {
  // deal with this other kind of exception
}
```

Note that the order in which the `catch` clauses occur is relevant—they are tried in order, and the first clause where the current exception is in a subclass of the specified exception will be chosen.