Programming for Correctness

Goal: correct programs
- What is correct, anyway?
  - Now: defining correct behavior
  - Later: finding out what users really want
- How to ensure this?
  - How to make programs more likely to be correct?
  - How to keep them correct as they evolve?

Specifications
- A specification describes what a method/class/... is supposed to do
- (Some) goals:
  - Precise
  - Complete
  - Understandable by people
  - Checkable by machines
  - Hard to meet all these goals

Pre-/post-conditions
- One way to think about a method's specification is by a pair of
  - A precondition: what the method assumes is true when it starts
    - Eg. what values its arguments are allowed to have
  - A postcondition: what the method guarantees is true when it returns
    - Eg. what the value it returns will be
    - Under the assumption that its precondition is met!

Examples
- double sqrt(double x):
  - pre: x >= 0
  - post:
    - result * result = x
    - result >= 0
- void sort(int[] values):
  - pre: values != null
  - post: for all i, j in [0..values-length):
    - if i < j then values[i] <= values[j]
    - (or, post: values is sorted in non-decreasing order)

Who's responsible?
- Preconditions are the responsibility of the caller
  - The callee method can assume they're true on entry
- Postconditions are the responsibility of the callee
  - The caller can assume they're true when the call returns
Fail-soft vs. fail-stop

- What happens if there's a bug in the program, and a pre- or post-condition isn't satisfied?
  - Things might still work, sort of
  - Eventually things might fail, but often in a bizarre way
  - Particularly true in "unsafe" languages like C, where violating a specification could cause unrelated memory to get corrupted
- Would like a cleaner failure, the moment the violation happens

Enforcement

- Can use various language and programming techniques to check pre- and post-conditions
  - Typically assume each pre- and post-condition is a regular boolean expression
  - Some languages have support for pre- and post-conditions built-in
    - Checked automatically on entry & exit
  - Others support *assertions*

Assertions

- An assertion is a boolean expression at a given point in the program that's checked at run-time
  - The expression should be true
    - If it's not, then the assertion has failed, and some sort of fatal error should be reported
  - Precondition ⇒ an assertion on entry to the method
  - Postcondition ⇒ an assertion at every return point of the method
  - What about exception throws?

Assertions in Java

- Java 1.4 has built-in support for assertions
  - A new kind of statement:
    - `assert booleanExpr : errorMsg ;`
  - Semantics:
    - Evaluate `booleanExpr`
    - If it's true, OK
    - If it's false, throw an `AssertionError`, which if unhandled will print out `errorMsg`

Example

```java
public void sort(int[] values) {
    assert values != null : "null argument";
    // the sorting algorithm here
    assert isSorted(values) : "sort broke!";
}
private boolean isSorted(int[] values) {
    // return whether values is sorted
}
```

Compiling & running with assertions

- To enable the `assert` statement, must invoke `javac` with the `-source 1.4` option
  - `javac -source 1.4 Main.java`...
- To run with assertion checking turned on, must invoke `java` with the `-ea` ("enable assertions") flag
  - `java -ea Main ...`
Disabling assertion checking
- Assertion checking can be expensive
- Often, assertion checking can be enabled or disabled, either at compile-time or at run-time
- Can have lots of assertions enabled during debugging, fewer during "normal" execution
- Can sometimes choose which class of assertions to enable, based on what part of the system needs extra checking

Assertions vs. error checking
- Don't use assertions to do regular error checking that should always be present
  - E.g. checking whether user input is OK
- Your program should still work, and do all necessary error checking, with assertions disabled

Specified errors
- A public library method often specifies what it does in all cases, including "error" cases
  - E.g., what exceptions are thrown for which kinds of "bad" inputs
- These error cases are not precondition assumptions, but are postcondition guarantees
  - Don't use assertions for them!
- Good style for public library methods to have no preconditions but instead to specify a response (e.g. an exception) for all possible inputs

Example
- double sqrt(double x):
  - post:
    - if \( x \geq 0 \):
      - result \* result = x
      - result \geq 0
    - otherwise:
      - throws IllegalArgumentException

Invariants
- A very useful kind of "specification" is an invariant
  - Something that is always true about some part of the software
- A great mental tool in thinking about the correctness of complex algorithms & data structures
- A great debugging tool, also

Simple invariants
- One kind of invariant is something that's true at some point in the program
  - If it's not true, then something broke
- An assertion is great for making such invariants explicit
  - E.g. in the middle of the sorting loop, all values in the array at indexes \( \leq i \) have been sorted
  - A loop invariant
Class invariants

- A class invariant is true about the state of each instance of the class
  - Established by the constructor
  - Preserved by all public methods
    - Can be temporarily violated in the middle of a modification
  - E.g., a binary search tree is always properly sorted
  - Can be viewed as implicit postconditions of all constructors and public methods

Formality

- These pre- & post-conditions are pretty formal
  - Makes them precise, processable by machine
  - Mostly clear to humans, for these examples
- As functions get more complex, it’s increasingly hard to be formal
  - Specifications get very long & involved
  - They become less readable by humans
- Informal specifications, even partial specifications, are better than no specifications!

Documentation

- The documentation is the main "specification" most people use
  - The more precise, the better
- Several tools can derive documentation from source code
  - E.g. javadoc, which produces web pages
    - Tools for special /** */ comments
  - Documentation in source code is less likely to be out of date
  - But anything that’s not machine-checked can get out of date

Literate programming

- Literate programming: code is just a part of an enclosing document
  - The document is primary, not the code
  - Like any technical document, can have examples, diagrams, references, etc.
  - Encourages good explanations, documentation
  - See e.g. noweb