Solutions to Exercises 6

6.1.1 Programming a trigonometric "package" in C:

(a) Write a header file `trig.h` (which corresponds roughly to the ADA package specification):

```
float sin (float x);
float cos (float x);
```

Also write a compilation unit `trig.c` (which corresponds roughly to the ADA package specification):

```
#include "trig.h"
#define twice_pi 6.2832
float norm (float x) {
   …  /* compute x module twice_pi */
}
float sin (float x) {
   …  /* compute sine of norm(x) */
}
float cos (float x) {
   …  /* compute cosine of norm(x) */
}
```

(b) Disadvantages:

- `twice_pi` and `norm` are not private
- calls to `sin` and `cos` are not fully type-checked.

6.2.2 `Complex` abstract type in ADA:

(a) Possible design:

```
package Complex_Numbers is
   type Complex is private;
   zero, one, j: constant Complex;
   function make (re, im: Float) return Complex;
   function polar (r, th: Float) return Complex;
   function magnitude (c: Complex) return Float;
   function "+" (c1, c2: Complex) return Complex;
   function "-" (c1, c2: Complex) return Complex;
   function "*" (c1, c2: Complex) return Complex;
private
   …
end Complex_Numbers;
```

(b) A possible representation would be a record whose fields are the complex number's real and imaginary parts. An alternative representation would use polar coordinates.

6.2.3 `Date` abstract type in ADA:

(a) Possible design:

```
package Dates is
   type Date is limited private;
   function make (y: Year_Number;
                  m: Month_Number;
                  d: Day_Number) return Date;
   function "+" (d: Date, i: Integer) return Date;
   function "-" (d: Date, i: Integer) return Date;
   function "=" (d1, d2: Date) return Boolean;
   function "<" (d1, d2: Date) return Boolean;
   function image (d: Date) return String;
private
   …
end Dates;
```

(b) The best representation would be a single integer (interpreted as the number of days since 2000-01-01, say). An alternative but inferior representation would a record whose fields are the year, month, and day numbers.

* **6.2.5** `Rational` abstract type in ADA:

(a) Possible design:

```
package Rationals is
   type Rat is limited private;
   function "/" (m, n: Integer) return Rat;
   function "+" (r1, r2: Rat) return Rat;
   function "-" (r1, r2: Rat) return Rat;
   function "*" (r1, r2: Rat) return Rat;
   function "=" (r1, r2: Rat) return Boolean;
   function "<" (r1, r2: Rat) return Boolean;
private
   …
end Rationals;
```

(b) Representation by a pair of integers (*num*, *den*):

```
package Rationals is
   …
private
   type Rat is record
          num: Integer;
          den: Positive;
       end record;
end Rationals;
package Rationals is
   function "/" (m, n: Integer) return Rat is
   begin
      if n = 0 then
         raise constraint_error;
      elsif n < 0 then
         return (-m, -n);
      else
         return (m, n);
      end if;
   end;
```

```
    function "+" (r1, r2: Rat) return Rat is
    begin
      return (r1.num*r2.den + r2.num*r1.den,
                    r1.den*r2.den);
    end;
    function "-" (r1, r2: Rat) return Rat is
    begin
      return (r1.num*r2.den - r2.num*r1.den,
                    r1.den*r2.den);
    end;
    function "*" (r1, r2: Rat) return Rat is
    begin
      return (r1.num*r2.num, r1.den*r2.den);
    end;
    function "=" (r1, r2: Rat) return Boolean is
    begin
      return (r1.num*r2.den = r2.num*r1.den);
    end;
    function "<" (r1, r2: Rat) return Boolean is
    begin
      return (r1.num*r2.den < r2.num*r1.den);
    end;
  end Rationals;
```

Note that the `"="` operation must be redefined here, otherwise "`3/4 = 6/8`" would yield the wrong result.

(c) Representation by a pair of integers (*num*, *den*) with no common factor:

```
  package Rationals is
    …
  private
    type Rat is record
          num: Integer;    -- where num and den have
          den: Positive;   -- no common factor
        end record;
  end Rationals;
  package Rationals is
    function "/" (m, n: Integer) return Rat is
    begin
      if n = 0 then
        raise constraint_error;
      elsif n < 0 then
        return reduced(-m, -n);
      else
        return reduced(m, n);
      end if;
    end;
    function "+" (r1, r2: Rat) return Rat is
    begin
      return reduced(
          r1.num*r2.den + r2.num*r1.den,
                r1.den*r2.den);
    end;
```

```
function "-" (r1, r2: Rat) return Rat is
begin
  return reduced(
       r1.num*r2.den - r2.num*r1.den,
              r1.den*r2.den);
end;
function "*" (r1, r2: Rat) return Rat is
begin
  return reduced(
       r1.num*r2.num, r1.den*r2.den);
end;
function "=" (r1, r2: Rat) return Boolean is
begin
  return (r1.num = r2.num
       and then r1.den = r1.den);
end;
function "<" (r1, r2: Rat) return Boolean is
begin
  return (r1.num*r2.den < r2.num*r1.den);
end;
function reduced (m, n: Integer) return Rat is
  f: Positive := gcd(m, n);
begin
  return (m/f, n/f);
end;
end Rationals;
```

Advantages and disadvantages of this representation:

- \+ risk of overflow is reduced
- – most operations are slower (but `"="` is faster).

6.3.1 `Counter` class in JAVA:

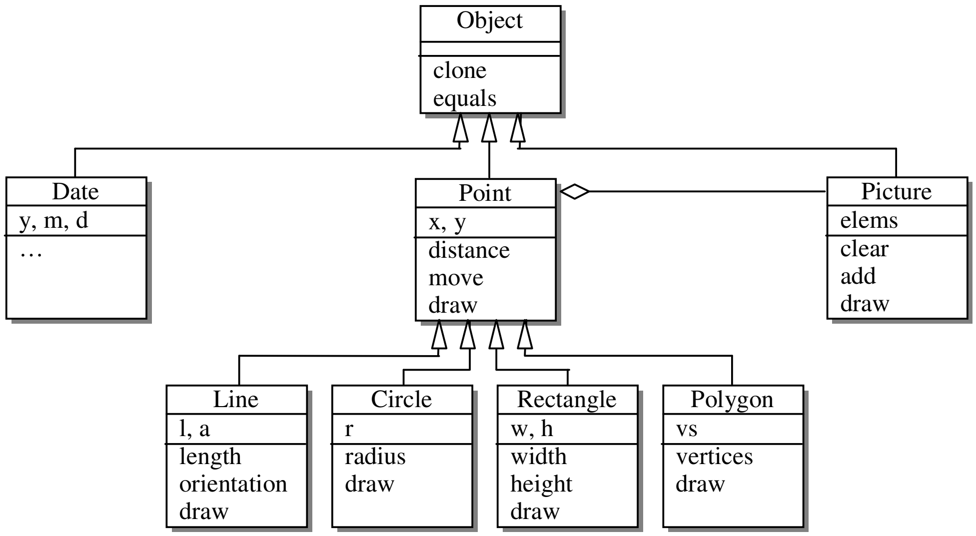
(a) Possible design:

```
class Counter {
  private …;
  public Counter () { … }
  public void zero () { … }
  public void inc () { … }
  public int get () { … }
}
```

(b) Implementation:

```
class Counter {
  private int n;
  public Counter () { n = 0; }
  public void zero () { n = 0; }
  public void inc () { n++; }
  public int get () { return n; }
}
```

6.3.2 Class hierarchy:



* **6.3.3** `Text` class in JAVA:

(a) Possible design:

```
class Text {
  private …;
  public Text () { … }
  public void clear () { … }
  public void insert (int pos, Text t) { … }
  public Text delete (int pos1, int pos2) { … }
  public void load (String filename) { … }
  public void save (String filename) { … }
  public void display (int lineWidth) { … }
}
```

(b) The simplest representation of a text would be a long array of characters, using a suitable control character (say `'\n'`) to separate paragraphs:

```
private char[] cs;
```

6.3.4 `Rational` class in JAVA:

(a) Possible design:

```
class Rational {
  private …;
  public Rational (int m, int n) { … }
  public Rational plus (Rational r) { … }
  public Rational minus (Rational r) { … }
  public Rational times (Rational r) { … }
  public boolean equals (Rational r) { … }
  public int compareTo (Rational r) { … }
}
```

(b) Representation by a pair of integers (*num*, *den*):

```
class Rational {
  private int num, den;  // den > 0
  public Rational (int m, int n) {
    if (n = 0) throw …;
    else if (n < 0) { num = -m; den = -n; }
    else { num = m; den = n; }
  }
  public Rational plus (Rational r) {
    this.num = this.num*r.den + r.num*this.den;
    this.num = this.den*r.den;
  }
  public Rational minus (Rational r) { … }
  public Rational times (Rational r) { … }
  public boolean equals (Rational r) {
    return (this.num*r.den == r.num*this.den);
  }
  public int compareTo (Rational r) {
    return this.num*r.den - r.num*this.den;
  }
}
```

6.4.1 Representation of objects of the classes of Exercises 6.3.1–4: