Chapter 1

Concurrency

1.1 Data races

A *data race* happens when there are two memory accesses that

- target the same location, for example, the same static field,
- are performed concurrently by different threads,
- are not both reads, that is, at least one is a write, and
- are not synchronized, for example, by a synchronized block.

A data race, although in itself not a bug, is often an indication of potential presence of a bug. Therefore, detecting data races in concurrent Java code is valuable. As we will see, JPF can detect data races.

Consider the following `Account` class.

```
public class Account {
 private int balance;

 public Account() {
  this.balance = 0;
 }

 public void deposit(int amount) {
  this.balance += amount;
 }

 public int getBalance() {
  return this.balance;
 }
}
```

Note that the method `deposit` is not synchronized. A `Customer` is a `Thread` that has an `Account` to which one dollar is deposited.

```
public class Customer extends Thread {
 private Account account;

 public Customer(Account account) {
  this.account = account;
  }

  public void run() {
    this.account.deposit(1);
  }
}
```

Assume that two `Customers` deposit concurrently.

```
public class Customers {
  public static void main(String[] args) {
    Account account = new Account();
    Customer first = new Customer(account);
    Customer second = new Customer(account);
    first.start();
    second.start();
  }
}
```

Then we may encounter a data race on the field `balance`. To detect this data race by JPF, we use the following application properties file.

```
target = Customers
listener = gov.nasa.jpf.listener.PreciseRaceDetector
```

When we run JPF with the above application properties file we obtain the following error.

```
====================================================== error 1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Account@162.balance
  Thread-1 at Account.deposit(Account.java:9)
" WRITE: putfield Account.balance
  Thread-2 at Account.deposit(Account.java:9)
" READ: getfield Account.balance
```

Both threads execute line 9 of the `Account` class concurrently. Recall that `this.balance += amount` is not atomic and consists of three bytecode instructions that read the value of the field (getfield), increment its value, and write the new value of the field (putfield). According to the above error, the first thread can write its new value of the field while the second thread reads the value of the field concurrently.

By default, JPF terminates after detecting the first error. By setting the property `search.multiple_errors` to true, JPF finds several other data races (for example, one where both threads write the new value concurrently).

In this case, the data race leads us to a bug. For example, let us expand our app as follows.

```
1  public class Customers {
2    public static void main(String[] args) throws InterruptedException {
3      Account account = new Account();
4      Customer first = new Customer(account);
5      Customer second = new Customer(account);
6      first.start();
7      second.start();
8      first.join();
9      second.join();
10     assert account.getBalance == 2 : "" + account.getBalance();
11   }
12 }
```

Then JPF reports that the assert at line 10 is violated.

```
1 ====================================================== error 1
2 gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
3 java.lang.AssertionError: 1
4       at Customers.main(Customers.java:10)
```

When we reach line 10, the balance of the account may be 1 instead of 2, as reported in line 3. This bug can be fixed by declaring the `deposit` method as synchronized.

The listener `PreciseRaceDetector` can be customized by means of the following two properties:

- The property `race.exclude` contains those packages that are not considered when checking for data races. By default, the standard libraries, that is, `java.*` and `javax.*` are excluded.
- If set, the property `race.include` contains those packages that are considered when checking for data races. By default, this property is not set and, as a consequence, all packages, apart from those excluded by `race.exclude`, are considered.