Doug Lea

Concurrent Programming in Java™ Second Edition

Design Principles and Patterns

The Java™ Series

... from the Source™

Sun microsystems

JAVA

The members of Ralph Johnson's patterns seminar (especially Brian Foote and Ian Chai) read through early forms of some patterns and suggested many improvements. Raj Datta, Sterling Barrett, and Philip Eskelin of the New York City Patterns Group, and Russ Rufer, Ming Kwok, Mustafa Ozgen, Edward Anderson, and Don Chin of the Silicon Valley Patterns Group performed similar valuable service for preliminary versions of the second edition.

Official and unofficial reviewers of the first- and second-edition manuscripts made helpful comments and suggestions on tight schedules. They include Ken Arnold, Josh Bloch, Joseph Bowbeer, Patrick Chan, Gary Craig, Desmond D'Souza, Bill Foote, Tim Harrison, David Henderson, Tim Lindholm, Tom May, Oscar Nierstrasz, James Robins, Greg Travis, Mark Wales, Peter Welch, and Deborra Zukowski. Very special thanks go to Tom Cargill for his many insights and corrections, as well as for permission to include a description of his Specific Notification pattern. Very special thanks also go to David Holmes for, among many contributions, helping to develop and extend material for tutorials that in turn became included in the second edition.

Rosemary Simpson contributed numerous improvements in the course of creating the index. Ken Arnold patiently helped me deal with FrameMaker. Mike Hendrickson and the editorial crew at Addison-Wesley have been continually supportive.

This book would not have been possible without the generous support of Sun Labs. Thanks especially to Jos Marlowe and Steve Heller for providing opportunities to work collaboratively on fun and exciting research and development projects.

Thanks above all to Kathy, Keith, and Colin for tolerating all this.

Doug Lea, September, 1999

Chapter 1. Concurrent Object-Oriented Programming

This book discusses some ways of thinking about, designing, and implementing concurrent programs in the Java™ programming language. Most presentations in this book assume that you are an experienced developer familiar with object-oriented (OO) programming, but have little exposure to concurrency. Readers with the opposite background — experience with concurrency in other languages — may also find this book useful.

The book is organized into four coarse-grained chapters. (Perhaps *parts* would be a better term.) This first chapter begins with a brief tour of some frequently used constructs and then backs up to establish a conceptual basis for concurrent object-oriented programming: how concurrency and objects fit together, how the resulting design forces impact construction of classes and components, and how some common design patterns can be used to structure solutions.

The three subsequent chapters are centered around use (and evasion) of the three kinds of concurrency constructs found in the Java programming language:

Exclusion. Maintaining consistent states of objects by preventing unwanted interference among concurrent activities, often using `synchronized` methods.

State dependence. Triggering, preventing, postponing, or recovering from actions depending on whether objects are in states in which these actions could or did succeed, sometimes using *monitor* methods `Object.wait`, `Object.notify`, and `Object.notifyAll`.

Creating threads. Establishing and managing concurrency, using `Thread` objects.

Each chapter contains a sequence of major sections, each on an independent topic. They present high-level design principles and strategies, technical details surrounding constructs, utilities that encapsulate common usages, and associated design patterns that address particular concurrency problems. Most sections conclude with an annotated set of further readings providing more information on selected topics. The online supplement to this book contains links to additional online resources, as well as updates, errata, and code examples. It is accessible via links from:

http://java.sun.com/Series or http://gee.cs.oswego.edu/dl/cpj

If you are already familiar with the basics, you can read this book in the presented order to explore each topic in more depth. But most readers will want to read this book in various different orders. Because most concurrency concepts and techniques interact with most others, it is not always possible to understand each section or chapter in complete isolation from all the others. However, you can still take a breadth-first approach, briefly scanning each chapter (including this one) before proceeding with more detailed coverage of interest. Many presentations later in the book can be approached after selectively reading through earlier material indicated by extensive cross-references.

You can practice this now by skimming through the following preliminaries.

Terminology. This book uses standard OO terminological conventions: programs define *methods* (implementing *operations*) and *fields* (representing *attributes*) that hold for all *instances* (objects) of specified *classes.*

Interactions in OO programs normally revolve around the responsibilities placed upon a *client* object needing an action to be performed, and a *server* object containing the code to perform the action. The terms *client* and *server* are used here in their generic senses, not in the specialized sense of distributed client/server architectures. A client is just any object that sends a request to another object, and a server is just any object receiving such a request. Most objects play the roles of both clients and servers. In the usual case where it doesn't matter whether an object under discussion acts as a client or server or both, it is usually called a *host;* others that it may in turn interact with are often called *helpers* or *peers.* Also, when discussing invocations of the form `obj.msg(arg)`, the recipient (that is, the object bound to variable `obj`) is called the *target* object.

This book generally avoids dealing with transient facts about particular classes and packages not directly related to concurrency. And it does *not* cover details about concurrency control in specialized frameworks such as Enterprise JavaBeans™ and Servlets. But it does sometimes refer to branded software and trademarked products associated with the Java™ Platform. The copyright page of this book provides more information.

Code listings. Most techniques and patterns in this book are illustrated by variants of an annoyingly small set of toy running examples. This is not an effort to be boring, but to be clear. Concurrency constructs are often subtle enough to get lost in otherwise meaningful examples. Reuse of running examples makes small but critical differences more obvious by highlighting the main design and implementation issues. Also, the presentations include code sketches and fragments of classes that illustrate implementation techniques, but are not intended to be complete or even compilable. These classes are indicated by leading comments in the listings.

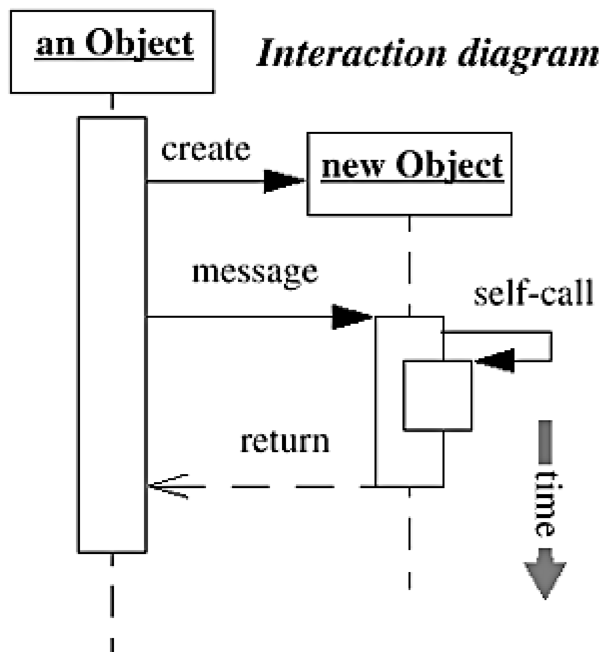
Import statements, access qualifiers, and even methods and fields are sometimes omitted from listings when they can be inferred from context or do not impact relevant functionality. The `protected`

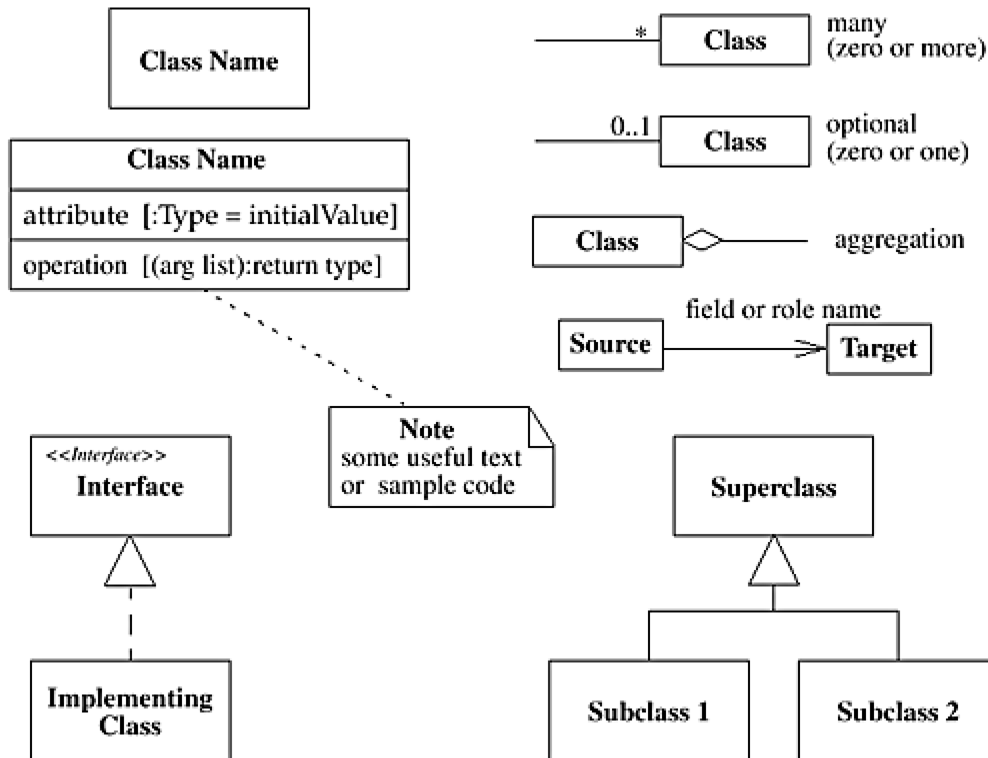
qualifier is used as a default for non-public features whenever there is no particular reason to restrict subclass access. This emphasizes opportunities for extensibility in concurrent class design (see § 1.3.4 and § 3.3.3). Classes by default have no access qualifier. Sample listings are sometimes formatted in nonstandard ways to keep them together on pages or to emphasize the main constructions of interest.

The code for all example classes in this book is available from the online supplement. Most techniques and patterns in this book are illustrated by a single code example showing their most typical forms. The supplement includes additional examples that demonstrate minor variations, as well as some links to other known usages. It also includes some larger examples that are more useful to browse and experiment with online than to read as listings.

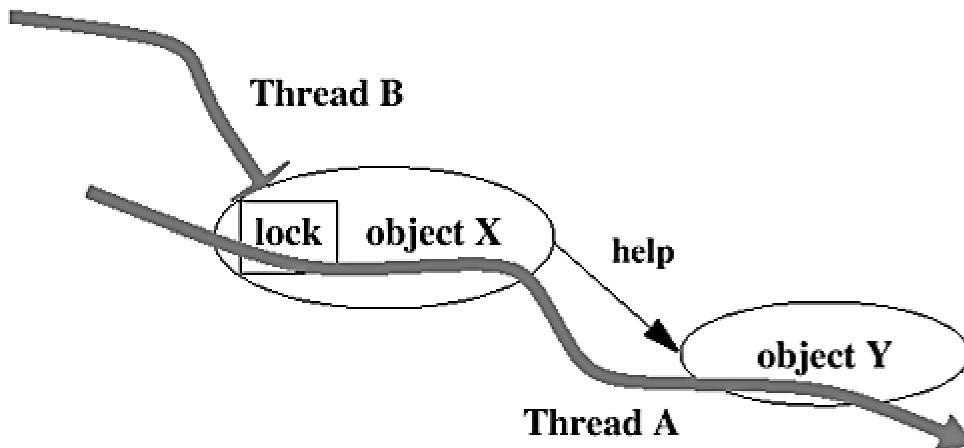
The supplement provides links to a package, `util.concurrent`, that contains production-quality versions of utility classes discussed in this book. This code runs on the Java 2 Platform and has been tested with 1.2.x releases. Occasional discussions, asides, and footnotes briefly mention changes from previous releases, potential future changes known at the time of this writing, and a few implementation quirks to watch out for. Check the online supplement for additional updates.

Diagrams. Standard UML notation is used for interaction and class diagrams (see the Further Readings in § 1.1.3). The accompanying diagrams (courtesy of Martin Fowler) illustrate the only forms used in this book. Other aspects of UML notation, methodology, and terminology are not specifically relied on.





Most other diagrams show *timethreads* in which free-form gray curves trace threads traversing through collections of objects. Flattened arrowheads represent blocking. Objects are depicted as ovals that sometimes show selected internal features such as locks, fields, and bits of code. Thin (usually labeled) lines between objects represent relations (normally references or potential calls) between them. Here's an otherwise meaningless example showing that thread A has acquired the lock for object X, and is proceeding through some method in object Y that serves as a helper to X. Thread B is meanwhile somehow blocked while entering some method in object X:



1.1 Using Concurrency Constructs

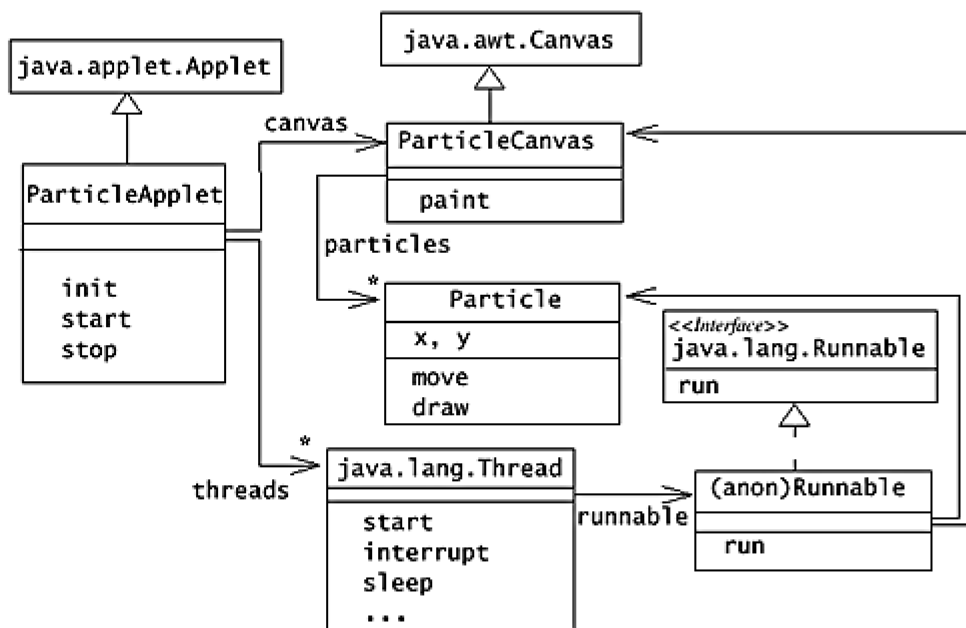
This section introduces basic concurrency support constructs by example and then proceeds with a walk-through of the principal methods of class `Thread`. Other concurrency constructs are briefly

described as they are introduced, but full technical details are postponed to later chapters (mainly § 2.2.1 and § 3.2.2). Also, concurrent programs often make use of a few ordinary Java programming language features that are not as widely used elsewhere. These are briefly reviewed as they arise.

1.1.1 A Particle Applet

`ParticleApplet` is an `Applet` that displays randomly moving particles. In addition to concurrency constructs, this example illustrates a few of the issues encountered when using threads with any GUI-based program. The version described here needs a lot of embellishment to be visually attractive or realistic. You might enjoy experimenting with additions and variations as an exercise.

As is typical of GUI-based programs, `ParticleApplet` uses several auxiliary classes that do most of the work. We'll step through construction of the `Particle` and `ParticleCanvas` classes before discussing `ParticleApplet`.



1.1.1.1 Particle

The `Particle` class defines a completely unrealistic model of movable bodies. Each particle is represented only by its (x, y) location. Each particle also supports a method to randomly change its location and a method to draw itself (as a small square) given a supplied `java.awt.Graphics` object.

While `Particle` objects do not themselves exhibit any intrinsic concurrency, their methods may be invoked across multiple concurrent activities. When one activity is performing a `move` and another is invoking `draw` at about the same time, we'd like to make sure that the `draw` paints an accurate representation of where the `Particle` is. Here, we require that `draw` uses the location values current either *before* or *after* the move. For example, it would be conceptually wrong for a `draw` operation to display using the y-value current before a given move, but the x-value current after the move. If we were to allow this, then the `draw` method would sometimes display the particle at a location that it never actually occupied.

This protection can be obtained using the `synchronized` keyword, which can modify either a method or a block of code. *Every instance* of class `Object` (and its subclasses) possesses a lock that is obtained on entry to a `synchronized` method and automatically released upon exit. The code-block version works in the same way except that it takes an argument stating which object to lock. The most common argument is `this`, meaning to lock the object whose method is executing. When a lock is held by one thread, other threads must block waiting for the holding thread to release the lock. Locking has no effect on non-synchronized methods, which can execute even if the lock is being held by another thread.

Locking provides protection against both high-level and low-level conflicts by enforcing *atomicity* among methods and code-blocks `synchronized` on the same object. Atomic actions are performed as units, without any interleaving of the actions of other threads. But, as discussed in § 1.3.2 and in Chapter 2, too much locking can also produce liveness problems that cause programs to freeze up. Rather than exploring these issues in detail now, we'll rely on some simple default rules for writing methods that preclude interference problems:

- **Always lock during updates to object fields.**
- **Always lock during access of possibly updated object fields.**
- **Never lock when invoking methods on other objects.**

These rules have many exceptions and refinements, but they provide enough guidance to write class `Particle`:

```
import java.util.Random;

class Particle {
 protected int x;
 protected int y;
 protected final Random rng = new Random();

 public Particle(int initialX, int initialY) {
  x = initialX;
  y = initialY;
 }

 public synchronized void move() {
  x += rng.nextInt(10) - 5;
  y += rng.nextInt(20) - 10;
 }

 public void draw(Graphics g) {
  int lx, ly;
  synchronized (this) { lx = x; ly = y; }
  g.drawRect(lx, ly, 10, 10);
 }
}
```

Notes:

- The use of `final` in the declaration of the random number generator `rng` reflects our decision that this reference field cannot be changed, so it is not impacted by our locking rules. Many concurrent programs use `final` extensively, in part as helpful, automatically enforced documentation of design decisions that reduce the need for synchronization (see § 2.1).
- The `draw` method needs to obtain a consistent snapshot of both the x and y values. Since a single method can return only one value at a time, and we need both the x and y values here, we cannot easily encapsulate the field accesses as a `synchronized` method. We instead use a `synchronized` block. (See § 2.4 for some alternatives.)
- The `draw` method conforms to our rule of thumb to release locks during method invocations on other objects (here `g.drawRect`). The `move` method appears to break this rule by calling `rng.nextInt`. However, this is a reasonable choice here because each `Particle` *confines* its own `rng` — conceptually, the `rng` is just a part of the `Particle` itself, so it doesn't count as an "other" object in the rule. Section § 2.3 describes more general conditions under which this sort of reasoning applies and discusses factors that should be taken into account to be sure that this decision is warranted.

1.1.1.2 ParticleCanvas

`ParticleCanvas` is a simple subclass of `java.awt.Canvas` that provides a drawing area for all of the `Particles`. Its main responsibility is to invoke `draw` for all existing particles whenever its `paint` method is called.

However, the `ParticleCanvas` itself does not create or manage the particles. It needs either to be told about them or to ask about them. Here, we choose the former.

The instance variable `particles` holds the array of existing `Particle` objects. This field is set when necessary by the applet, but is used in the `paint` method. We can again apply our default rules, which in this case lead to the creation of little `synchronized` `get` and `set` methods (also known as *accessor* and *assignment* methods) for `particles`, otherwise avoiding direct access of the `particles` variable itself. To simplify and to enforce proper usage, the `particles` field is never allowed to be `null`. It is instead initialized to an empty array:

```
class ParticleCanvas extends Canvas {

 private Particle[] particles = new Particle[0];

 ParticleCanvas(int size) {
  setSize(new Dimension(size, size));
 }

 // intended to be called by applet
 protected synchronized void setParticles(Particle[] ps) {
  if (ps == null)
    throw new IllegalArgumentException("Cannot set null");

  particles = ps;
 }
```

```
 protected synchronized Particle[] getParticles() {
  return particles;
 }

 public void paint(Graphics g) { // override Canvas.paint
  Particle[] ps = getParticles();

  for (int i = 0; i < ps.length; ++i)
   ps[i].draw(g);
 }
}
```

1.1.1.3 ParticleApplet

The `Particle` and `ParticleCanvas` classes could be used as the basis of several different programs. But in `ParticleApplet` all we want to do is set each of a collection of particles in autonomous "continuous" motion and update the display accordingly to show where they are. To comply with standard applet conventions, these activities should begin when `Applet.start` is externally invoked (normally from within a web browser), and should end when `Applet.stop` is invoked. (We could also add buttons allowing users to start and stop the particle animation themselves.)

There are several ways to implement all this. Among the simplest is to associate an independent loop with each particle and to run each looping action in a different thread.

Actions to be performed within new threads must be defined in classes implementing `java.lang.Runnable`. This interface lists only the single method `run`, taking no arguments, returning no results, and throwing no checked exceptions:

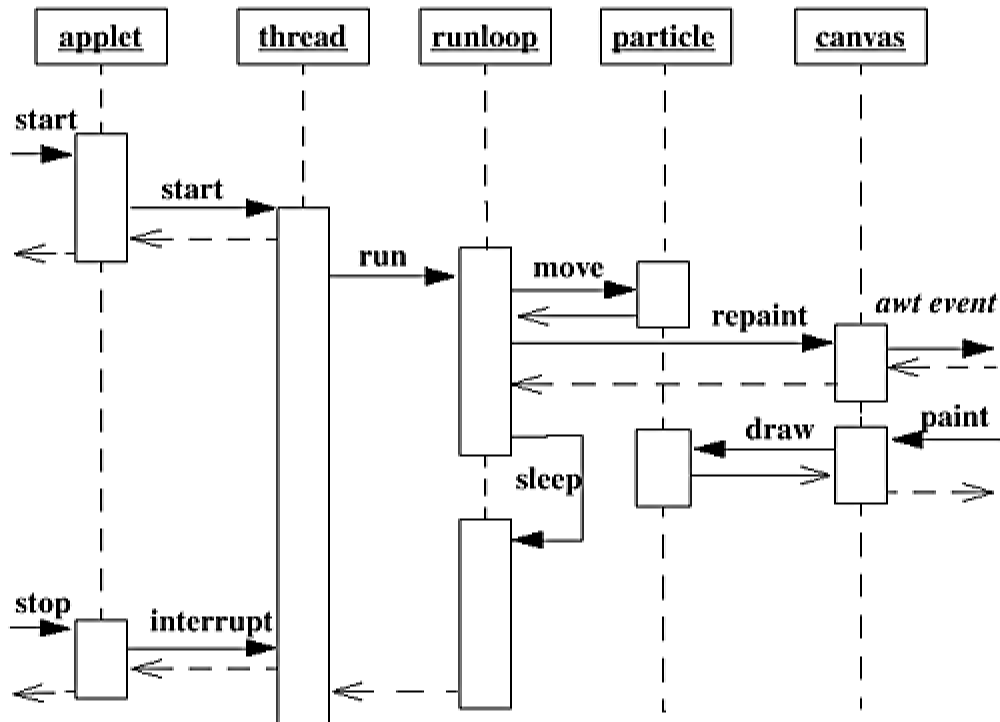
```
public interface java.lang.Runnable {
 void run();
}
```

An `interface` encapsulates a coherent set of services and attributes (broadly, a *role*) without assigning this functionality to any particular object or code. Interfaces are more abstract than classes since they say nothing at all about representations or code. All they do is describe the *signatures* (names, arguments, result types, and exceptions) of public operations, without even pinning down the classes of the objects that can perform them. The classes that can support `Runnable` typically have nothing in common except that they contain a `run` method.

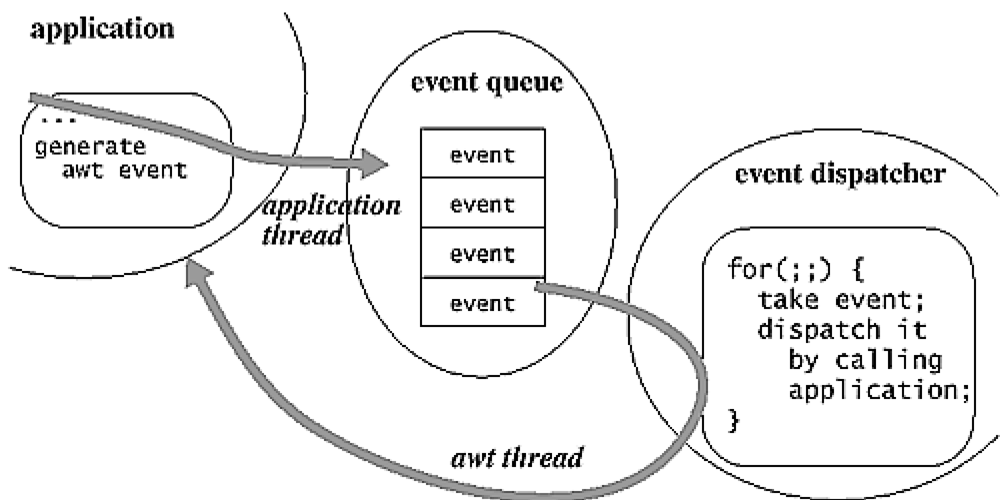
Each instance of the `Thread` class maintains the control state necessary to execute and manage the call sequence comprising its action. The most commonly used constructor in class `Thread` accepts a `Runnable` object as an argument, which arranges to invoke the `Runnable`'s `run` method when the thread is started. While any class can implement `Runnable`, it often turns out to be both convenient and helpful to define a `Runnable` as an anonymous inner class.

The `ParticleApplet` class uses threads in this way to put particles into motion, and cancels them when the applet is finished. This is done by overriding the standard `Applet` methods `start`

and `stop` (which have the same names as, but are unrelated to, methods `Thread.start` and `Thread.stop`).



The above interaction diagram shows the main message sequences during execution of the applet. In addition to the threads explicitly created, this applet interacts with the AWT event thread, described in more detail in § 4.1.4. The producer-consumer relationship extending from the omitted right hand side of the interaction diagram takes the approximate form:




```
public class ParticleApplet extends Applet {

 protected Thread[] threads = null; // null when not running
```

```
protected final ParticleCanvas canvas
                        = new ParticleCanvas(100);

public void init() { add(canvas); }

protected Thread makeThread(final Particle p) { // utility
 Runnable runloop = new Runnable() {
  public void run() {
   try {
     for(;;) {
      p.move();
      canvas.repaint();
      Thread.sleep(100); // 100msec is arbitrary
     }
    }
    catch (InterruptedException e) { return; }
   }
  };
  return new Thread(runloop);
 }

 public synchronized void start() {
  int n = 10; // just for demo

  if (threads == null) { // bypass if already started
   Particle[] particles = new Particle[n];
   for (int i = 0; i < n; ++i)
    particles[i] = new Particle(50, 50);
   canvas.setParticles(particles);

   threads = new Thread[n];
   for (int i = 0; i < n; ++i) {
    threads[i] = makeThread(particles[i]);
    threads[i].start();
   }
  }
 }

 public synchronized void stop() {
  if (threads != null) { // bypass if already stopped
    for (int i = 0; i < threads.length; ++i)
     threads[i].interrupt();
    threads = null;
   }
 }
}
```

Notes:

- The action in `makeThread` defines a "forever" loop (which some people prefer to write equivalently as "`while (true)`") that is broken only when the current thread is interrupted. During each iteration, the particle moves, tells the canvas to repaint so the move will be displayed, and then does nothing for a while, to slow things down to a human-viewable rate. `Thread.sleep` pauses the current thread. It is later resumed by a system timer.
- One reason that inner classes are convenient and useful is that they *capture* all appropriate context variables — here `p` and `canvas` — without the need to create a separate class with fields that record these values. This convenience comes at the price of one minor awkwardness: All captured method arguments and local variables must be declared as `final`, as a guarantee that the values can indeed be captured unambiguously. Otherwise, for example, if `p` were reassigned after constructing the `Runnable` inside method `makeThread`, then it would be ambiguous whether to use the original or the assigned value when executing the `Runnable`.
- The call to `canvas.repaint` does not directly invoke `canvas.paint`. The `repaint` method instead places an `UpdateEvent` on a `java.awt.EventQueue`. (This may be internally optimized and further manipulated to eliminate duplicate events.) A `java.awt.EventDispatchThread` asynchronously takes this event from the queue and dispatches it by (ultimately) invoking `canvas.paint`. This thread and possibly other system-created threads may exist even in nominally single-threaded programs.
- The activity represented by a constructed `Thread` object does not begin until invocation of the `Thread.start` method.
- As discussed in § 3.1.2, there are several ways to cause a thread's activity to stop. The simplest is just to have the `run` method terminate normally. But in infinitely looping methods, the best option is to use `Thread.interrupt`. An interrupted thread will automatically abort (via an `InterruptedException`) from the methods `Object.wait`, `Thread.join`, and `Thread.sleep`. Callers can then catch this exception and take any appropriate action to shut down. Here, the `catch` in `runloop` just causes the `run` method to exit, which in turn causes the thread to terminate.
- The `start` and `stop` methods are `synchronized` to preclude concurrent starts or stops. Locking works out OK here even though these methods need to perform many operations (including calls to other objects) to achieve the required started-to-stopped or stopped-to-started state transitions. Nullness of variable `threads` is used as a convenient state indicator.

1.1.2 Thread Mechanics

A thread is a call sequence that executes independently of others, while at the same time possibly sharing underlying system resources such as files, as well as accessing other objects constructed within the same program (see § 1.2.2). A `java.lang.Thread` *object* maintains bookkeeping and control for this activity.

Every program consists of at least one thread — the one that runs the `main` method of the class provided as a startup argument to the Java virtual machine ("JVM"). Other internal background threads may also be started during JVM initialization. The number and nature of such threads vary across JVM implementations. However, all user-level threads are explicitly constructed and started from the main thread, or from any other threads that they in turn create.

Here is a summary of the principal methods and properties of class `Thread`, as well as a few usage notes. They are further discussed and illustrated throughout this book. *The Java™ Language*

Specification ("*JLS*") and the published API documentation should be consulted for more detailed and authoritative descriptions.

1.1.2.1 Construction

Different `Thread` constructors accept combinations of arguments supplying:

- A `Runnable` object, in which case a subsequent `Thread.start` invokes `run` of the supplied `Runnable` object. If no `Runnable` is supplied, the default implementation of `Thread.run` returns immediately.
- A `String` that serves as an identifier for the `Thread`. This can be useful for tracing and debugging, but plays no other role.
- The `ThreadGroup` in which the new `Thread` should be placed. If access to the `ThreadGroup` is not allowed, a `SecurityException` is thrown.

Class `Thread` itself implements `Runnable`. So, rather than supplying the code to be run in a `Runnable` and using it as an argument to a `Thread` constructor, you can create a subclass of `Thread` that overrides the `run` method to perform the desired actions. However, the best default strategy is to define a `Runnable` as a separate class and supply it in a `Thread` constructor. Isolating code within a distinct class relieves you of worrying about any potential interactions of `synchronized` methods or blocks used in the `Runnable` with any that may be used by methods of class `Thread`. More generally, this separation allows independent control over the nature of the action and the context in which it is run: The same `Runnable` can be supplied to threads that are otherwise initialized in different ways, as well as to other lightweight executors (see § 4.1.4). Also note that subclassing `Thread` precludes a class from subclassing any other class.

`Thread` objects also possess a daemon status attribute that cannot be set via any `Thread` constructor, but may be set only before a `Thread` is started. The method `setDaemon` asserts that the JVM may exit, abruptly terminating the thread, so long as all other non-daemon threads in the program have terminated. The `isDaemon` method returns status. The utility of daemon status is very limited. Even background threads often need to do some cleanup upon program exit. (The spelling of *daemon,* often pronounced as "day-mon", is a relic of systems programming tradition. System daemons are continuous processes, for example print-queue managers, that are "always" present on a system.)

1.1.2.2 Starting threads

Invoking its `start` method causes an instance of class `Thread` to initiate its `run` method as an independent activity. None of the synchronization locks held by the caller thread are held by the new thread (see § 2.2.1).

A `Thread` terminates when its `run` method completes by either returning normally or throwing an unchecked exception (i.e., `RuntimeException`, `Error`, or one of their subclasses). `Threads` are not restartable, even after they terminate. Invoking `start` more than once results in an `InvalidThreadStateException`.

The method `isAlive` returns `true` if a thread has been started but has not terminated. It will return `true` if the thread is merely blocked in some way. JVM implementations have been known to differ in the exact point at which `isAlive` returns `false` for threads that have been cancelled (see

§ 3.1.2). There is no method that tells you whether a thread that is not `isAlive` has ever been started. Also, one thread cannot readily determine which other thread started it, although it may determine the identities of other threads in its `ThreadGroup` (see § 1.1.2.6).

1.1.2.3 Priorities

To make it possible to implement the Java virtual machine across diverse hardware platforms and operating systems, the Java programming language makes no promises about scheduling or fairness, and does not even strictly guarantee that threads make forward progress (see § 3.4.1.5). But threads do support priority methods that heuristically influence schedulers:

- Each `Thread` has a priority, ranging between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY` (defined as `1` and `10` respectively).
- By default, each new thread has the same priority as the thread that created it. The initial thread associated with a `main` by default has priority `Thread.NORM_PRIORITY` (`5`).
- The current priority of any thread can be accessed via method `getPriority`.
- The priority of any thread can be dynamically changed via method `setPriority`. The maximum allowed priority for a thread is bounded by its `ThreadGroup`.

When more *runnable* (see § 1.3.2) threads than available CPUs, a scheduler is generally biased to prefer running those with higher priorities. The exact policy may and does vary across platforms. For example, some JVM implementations always select the thread with the highest current priority (with ties broken arbitrarily). Some JVM implementations map the ten `Thread` priorities into a smaller number of system-supported categories, so threads with different priorities may be treated equally. And some mix declared priorities with aging schemes or other scheduling policies to ensure that even low-priority threads eventually get a chance to run. Also, setting priorities may, but need not, affect scheduling with respect to other programs running on the same computer system.

Priorities have no other bearing on semantics or correctness (see § 1.3). In particular, priority manipulations cannot be used as a substitute for locking. Priorities can be used only to express the relative importance or urgency of different threads, where these priority indications would be useful to take into account when there is heavy contention among threads trying to get a chance to execute. For example, setting the priorities of the particle animation threads in `ParticleApplet` below that of the applet thread constructing them might on some systems improve responsiveness to mouse clicks, and would at least not hurt responsiveness on others. But programs should be designed to run correctly (although perhaps not as responsively) even if `setPriority` is defined as a no-op. (Similar remarks hold for `yield`; see § 1.1.2.5.)

The following table gives one set of general conventions for linking task categories to priority settings. In many concurrent applications, relatively few threads are actually runnable at any given time (others are all blocked in some way), in which case there is little reason to manipulate priorities. In other cases, minor tweaks in priority settings may play a small part in the final tuning of a concurrent system.

Range	Use
10	Crisis management
7-9	Interactive, event-driven
4-6	IO-bound
2-3	Background computation

1	Run only if nothing else can

1.1.2.4 Control methods

Only a few methods are available for communicating across threads:

- Each `Thread` has an associated boolean interruption status (see § 3.1.2). Invoking `t.interrupt` for some `Thread t` sets `t`'s interruption status to `true`, unless `Thread t` is engaged in `Object.wait`, `Thread.sleep`, or `Thread.join`; in this case `interrupt` causes these actions (in `t`) to throw `InterruptedException`, but `t`'s interruption status is set to `false`.
- The interruption status of any `Thread` can be inspected using method `isInterrupted`. This method returns `true` if the thread has been interrupted via the `interrupt` method but the status has not since been reset either by the thread invoking `Thread.interrupted` (see § 1.1.2.5) or in the course of `wait`, `sleep`, or `join` throwing `InterruptedException`.
- Invoking `t.join()` for `Thread t` suspends the *caller* until the target `Thread t` completes: the call to `t.join()` returns when `t.isAlive()` is `false` (see § 4.3.2). A version with a (millisecond) time argument returns control even if the thread has not completed within the specified time limit. Because of how `isAlive` is defined, it makes no sense to invoke `join` on a thread that has not been started. For similar reasons, it is unwise to try to `join` a `Thread` that you did not create.

Originally, class `Thread` supported the additional control methods `suspend`, `resume`, `stop`, and `destroy`. Methods `suspend`, `resume`, and `stop` have since been *deprecated;* method `destroy` has never been implemented in any release and probably never will be. The effects of methods `suspend` and `resume` can be obtained more safely and reliably using the waiting and notification techniques discussed in § 3.2. The problems surrounding `stop` are discussed in § 3.1.2.3.

1.1.2.5 Static methods

Some `Thread` class methods can be applied only to the thread that is currently running (i.e., the thread making the call to the `Thread` method). To enforce this, these methods are declared as `static`.

- `Thread.currentThread` returns a reference to the current `Thread`. This reference may then be used to invoke other (non-static) methods. For example, `Thread.currentThread().getPriority()` returns the priority of the thread making the call.
- `Thread.interrupted` clears interruption status of the current `Thread` and returns previous status. (Thus, one `Thread`'s interruption status cannot be cleared from other threads.)
- `Thread.sleep(long msecs)` causes the current thread to suspend for at least `msecs` milliseconds (see § 3.2.2).
- `Thread.yield` is a purely heuristic hint advising the JVM that if there are any other runnable but non-running threads, the scheduler should run one or more of these threads rather than the current thread. The JVM may interpret this hint in any way it likes.

Despite the lack of guarantees, `yield` can be pragmatically effective on some single-CPU JVM implementations that do not use time-sliced pre-emptive scheduling (see § 1.2.2). In this case, threads are rescheduled only when one blocks (for example on IO, or via `sleep`). On these systems, threads that perform time-consuming non-blocking computations can tie up a CPU for extended periods, decreasing the responsiveness of an application. As a safeguard, methods performing non-blocking computations that might exceed acceptable response times for event handlers or other reactive threads can insert `yields` (or perhaps even `sleeps`) and, when desirable, also run at lower priority settings. To minimize unnecessary impact, you can arrange to invoke `yield` only occasionally; for example, a loop might contain:

```
if (Math.random() < 0.01) Thread.yield();
```

On JVM implementations that employ pre-emptive scheduling policies, especially those on multiprocessors, it is possible and even desirable that the scheduler will simply ignore this hint provided by `yield`.

1.1.2.6 ThreadGroups

Every `Thread` is constructed as a member of a `ThreadGroup`, by default the same group as that of the `Thread` issuing the constructor for it. `ThreadGroups` nest in a tree-like fashion. When an object constructs a `new ThreadGroup`, it is nested under its current group. The method `getThreadGroup` returns the group of any thread. The `ThreadGroup` class in turn supports methods such as `enumerate` that indicate which threads are currently in the group.

One purpose of class `ThreadGroup` is to support security policies that dynamically restrict access to `Thread` operations; for example, to make it illegal to `interrupt` a thread that is not in your group. This is one part of a set of protective measures against problems that could occur, for example, if an applet were to try to kill the main screen display update thread. `ThreadGroups` may also place a ceiling on the maximum priority that any member thread can possess.

`ThreadGroups` tend not to be used directly in thread-based programs. In most applications, normal collection classes (for example `java.util.Vector`) are better choices for tracking groups of `Thread` objects for application-dependent purposes.

Among the few `ThreadGroup` methods that commonly come into play in concurrent programs is method `uncaughtException`, which is invoked when a thread in a group terminates due to an uncaught unchecked exception (for example a `NullPointerException`). This method normally causes a stack trace to be printed.

1.1.3 Further Readings

This book is not a reference manual on the Java programming language. (It is also not exclusively a how-to tutorial guide, or an academic textbook on concurrency, or a report on experimental research, or a book on design methodology or design patterns or pattern languages, but includes discussions on each of these facets of concurrency.) Most sections conclude with lists of resources that provide more information on selected topics. If you do a lot of concurrent programming, you will want to read more about some of them.

The *JLS* should be consulted for more authoritative accounts of the properties of Java programming language constructs summarized in this book:

Gosling, James, Bill Joy, and Guy Steele. *The Java™ Language Specification*, Addison-Wesley, 1996. As of this writing, a second edition of *JLS* is projected to contain clarifications and updates for the Java 2 Platform.

Introductory accounts include:

Arnold, Ken, and James Gosling. *The Java™ Programming Language, Second Edition*, Addison-Wesley, 1998.

If you have never written a program using threads, you may find it useful to work through either the online or book version of the *Threads* section of:

Campione, Mary, and Kathy Walrath. *The Java™ Tutorial, Second Edition*, Addison-Wesley, 1998.

A concise guide to UML notation is:

Fowler, Martin, with Kendall Scott. *UML Distilled, Second Edition*, Addison-Wesley, 1999. The UML diagram keys on pages 3-4 of the present book are excerpted by permission.

A more extensive account of UML is:

Rumbaugh, James, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

1.2 Objects and Concurrency

There are many ways to characterize objects, concurrency, and their relationships. This section discusses several different perspectives — definitional, system-based, stylistic, and modeling-based — that together help establish a conceptual basis for concurrent object-oriented programming.

1.2.1 Concurrency

Like most computing terms, "concurrency" is tricky to pin down. Informally, a concurrent program is one that does more than one thing at a time. For example, a web browser may be simultaneously performing an HTTP GET request to get an HTML page, playing an audio clip, displaying the number of bytes received of some image, and engaging in an advisory dialog with a user. However, this simultaneity is sometimes an illusion. On some computer systems these different activities might indeed be performed by different CPUs. But on other systems they are all performed by a single time-shared CPU that switches among different activities quickly enough that they appear to be simultaneous, or at least nondeterministically interleaved, to human observers.

A more precise, though not very interesting definition of concurrent programming can be phrased operationally: A Java virtual machine and its underlying operating system (OS) provide mappings from apparent simultaneity to physical parallelism (via multiple CPUs), or lack thereof, by allowing independent activities to proceed in parallel when possible and desirable, and otherwise by time-sharing. Concurrent programming consists of using programming constructs that are mapped in this way. Concurrent programming in the Java programming language entails using Java programming

language constructs to this effect, as opposed to system-level constructs that are used to create new operating system processes. By convention, this notion is further restricted to constructs affecting a single JVM, as opposed to distributed programming, for example using remote method invocation (RMI), that involves multiple JVMs residing on multiple computer systems.

Concurrency and the reasons for employing it are better captured by considering the nature of a few common types of concurrent applications:

Web services. Most socket-based web services (for example, HTTP daemons, servlet engines, and application servers) are multithreaded. Usually, the main motivation for supporting multiple concurrent connections is to ensure that new incoming connections do not need to wait out completion of others. This generally minimizes service latencies and improves availability.

Number crunching. Many computation-intensive tasks can be parallelized, and thus execute more quickly if multiple CPUs are present. Here the goal is to maximize throughput by exploiting parallelism.

I/O processing. Even on a nominally sequential computer, devices that perform reads and writes on disks, wires, etc., operate independently of the CPU. Concurrent programs can use the time otherwise wasted waiting for slow I/O, and can thus make more efficient use of a computer's resources.

Simulation. Concurrent programs can simulate physical objects with independent autonomous behaviors that are hard to capture in purely sequential programs.

GUI-based applications. Even though most user interfaces are intentionally single-threaded, they often establish or communicate with multithreaded services. Concurrency enables user controls to stay responsive even during time-consuming actions.

Component-based software. Large-granularity software components (for example those providing design tools such as layout editors) may internally construct threads in order to assist in bookkeeping, provide multimedia support, achieve greater autonomy, or improve performance.

Mobile code. Frameworks such as the `java.applet` package execute downloaded code in separate threads as one part of a set of policies that help to isolate, monitor, and control the effects of unknown code.

Embedded systems. Most programs running on small dedicated devices perform real-time control. Multiple components each continuously *react* to external inputs from sensors or other devices, and produce external outputs in a timely manner. As defined in *The Java™ Language Specification,* the Java platform does not support *hard* real-time control in which system correctness depends on actions being performed by certain deadlines. Particular run-time systems may provide the stronger guarantees required in some safety-critical hard-real-time systems. But all JVM implementations support *soft* real-time control, in which timeliness and performance are considered quality-of-service issues rather than correctness issues (see § 1.3.3). This reflects portability goals that enable the JVM to be implemented on modern opportunistic, multipurpose hardware and system software.

1.2.2 Concurrent Execution Constructs

Threads are only one of several constructs available for concurrently executing code. The idea of generating a new activity can be mapped to any of several abstractions along a granularity continuum reflecting trade-offs of autonomy versus overhead. Thread-based designs do not always provide the

best solution to a given concurrency problem. Selection of one of the alternatives discussed below can provide either more or less security, protection, fault-tolerance, and administrative control, with either more or less associated overhead. Differences among these options (and their associated programming support constructs) impact design strategies more than do any of the details surrounding each one.

1.2.2.1 Computer systems

If you had a large supply of computer systems, you might map each logical unit of execution to a different computer. Each computer system may be a uniprocessor, a multiprocessor, or even a cluster of machines administered as a single unit and sharing a common operating system. This provides unbounded autonomy and independence. Each system can be administered and controlled separately from all the others.

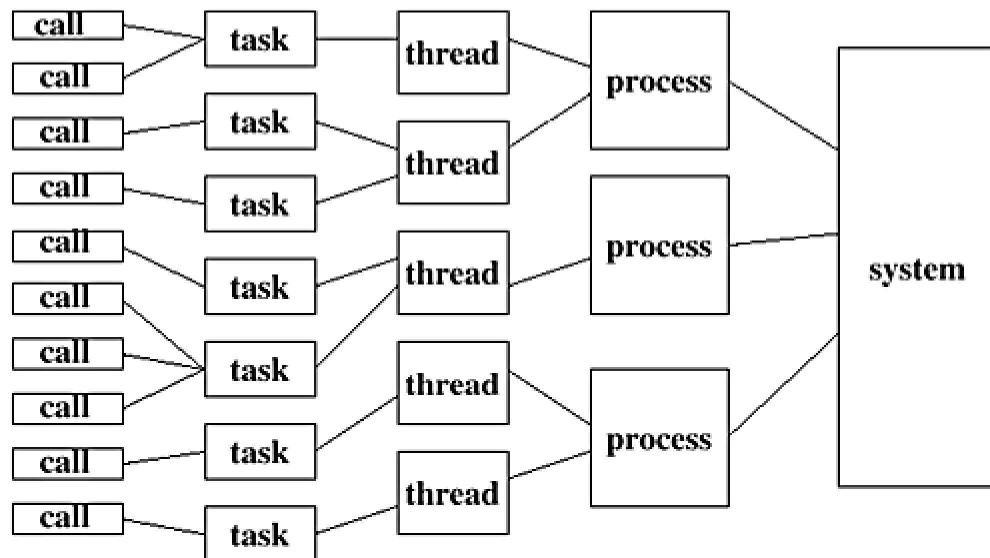
However, constructing, locating, reclaiming, and passing messages among such entities can be expensive, opportunities for sharing local resources are eliminated, and solutions to problems surrounding naming, security, fault-tolerance, recovery, and reachability are all relatively heavy in comparison with those seen in concurrent programs. So this mapping choice is typically applied only for those aspects of a system that intrinsically require a distributed solution. And even here, all but the tiniest embedded computer devices host more than one process.

1.2.2.2 Processes

A process is an operating-system abstraction that allows one computer system to support many units of execution. Each process typically represents a separate running program; for example, an executing JVM. Like the notion of a "computer system", a "process" is a logical abstraction, not a physical one. So, for example, bindings from processes to CPUs may vary dynamically.

Operating systems guarantee some degree of independence, lack of interference, and security among concurrently executing processes. Processes are generally not allowed to access one another's storage locations (although there are usually some exceptions), and must instead communicate via interprocess communication facilities such as pipes. Most systems make at least best-effort promises about how processes will be created and scheduled. This nearly always entails *pre-emptive* time-slicing — suspending processes on a periodic basis to give other processes a chance to run.

The overhead for creating, managing, and communicating among processes can be a lot lower than in per-machine solutions. However, since processes share underlying computational resources (CPUs, memory, IO channels, and so on), they are less autonomous. For example, a machine crash caused by one process kills all processes.



1.2.2.3 Threads

Thread constructs of various forms make further trade-offs in autonomy, in part for the sake of lower overhead. The main trade-offs are:

Sharing. Threads may share access to the memory, open files, and other resources associated with a single process. Threads in the Java programming language may share all such resources. Some operating systems also support intermediate constructions, for example "lightweight processes" and "kernel threads" that share only some resources, do so only upon explicit request, or impose other restrictions.

Scheduling. Independence guarantees may be weakened to support cheaper scheduling policies. At one extreme, all threads can be treated together as a single-threaded process, in which case they may *cooperatively* contend with each other so that only one thread is running at a time, without giving any other thread a chance to run until it blocks (see § 1.3.2). At the other extreme, the underlying scheduler can allow all threads in a system to contend directly with each other via pre-emptive scheduling rules. Threads in the Java programming language may be scheduled using any policy lying at or anywhere between these extremes.

Communication. Systems interact via communication across wires or wireless channels, for example using sockets. Processes may also communicate in this fashion, but may also use lighter mechanisms such as pipes and interprocess signalling facilities. Threads can use all of these options, plus other cheaper strategies relying on access to memory locations accessible across multiple threads, and employing memory-based synchronization facilities such as locks and waiting and notification mechanisms. These constructs support more efficient communication, but sometimes incur the expense of greater complexity and consequently greater potential for programming error.

1.2.2.4 Tasks and lightweight executable frameworks

The trade-offs made in supporting threads cover a wide range of applications, but are not always perfectly matched to the needs of a given activity. While performance details differ across platforms, the overhead in creating a thread is still significantly greater than the cheapest (but least independent) way to invoke a block of code — calling it directly in the current thread.

When thread creation and management overhead become performance concerns, you may be able to make additional trade-offs in autonomy by creating your own lighter-weight execution frameworks that impose further restrictions on usage (for example by forbidding use of certain forms of blocking), or make fewer scheduling guarantees, or restrict synchronization and communication to a more limited set of choices. As discussed in § 4.1.4, these tasks can then be mapped to threads in about the same way that threads are mapped to processes and computer systems.

The most familiar lightweight executable frameworks are event-based systems and subsystems (see § 1.2.3, § 3.6.4, and § 4.1), in which calls triggering conceptually asynchronous activities are maintained as events that may be queued and processed by background threads. Several additional lightweight executable frameworks are described in Chapter 4. When they apply, construction and use of such frameworks can improve both the structure and performance of concurrent programs. Their use reduces concerns (see § 1.3.3) that can otherwise inhibit the use of concurrent execution techniques for expressing logically asynchronous activities and logically autonomous objects (see § 1.2.4).

1.2.3 Concurrency and OO Programming

Objects and concurrency have been linked since the earliest days of each. The first concurrent OO programming language (created circa 1966), *Simula,* was also the first OO language, and was among the first concurrent languages. Simula's initial OO and concurrency constructs were somewhat primitive and awkward. For example, concurrency was based around *coroutines* — thread-like constructs requiring that programmers explicitly hand off control from one task to another. Several other languages providing both concurrency and OO constructs followed — indeed, even some of the earliest prototype versions of C++ included a few concurrency-support library classes. And Ada (although, in its first versions, scarcely an OO language) helped bring concurrent programming out from the world of specialized, low-level languages and systems.

OO design played no practical role in the multithreaded systems programming practices emerging in the 1970s. And concurrency played no practical role in the wide-scale embrace of OO programming that began in the 1980s. But interest in OO concurrency stayed alive in research laboratories and advanced development groups, and has re-emerged as an essential aspect of programming in part due to the popularity and ubiquity of the Java platform.

Concurrent OO programming shares most features with programming of any kind. But it differs in critical ways from the kinds of programming you may be most familiar with, as discussed below.

1.2.3.1 Sequential OO programming

Concurrent OO programs are often structured using the same programming techniques and design patterns as sequential OO programs (see for example § 1.4). But they are intrinsically more complex. When more than one activity can occur at a time, program execution is necessarily nondeterministic. Code may execute in surprising orders — any order that is not explicitly ruled out is allowed (see § 2.2.7). So you cannot always understand concurrent programs by sequentially reading through their code. For example, without further precautions, a field set to one value in one line of code may have a different value (due to the actions of some other concurrent activity) by the time the next line of code is executed. Dealing with this and other forms of *interference* often introduces the need for a bit more rigor and a more conservative outlook on design.

1.2.3.2 Event-based programming

Some concurrent programming techniques have much in common with those in event frameworks employed in GUI toolkits supported by `java.awt` and `javax.swing`, and in other languages such as Tcl/Tk and Visual Basic. In GUI frameworks, events such as mouse clicks are encapsulated as `Event` objects that are placed in a single `EventQueue`. These events are then dispatched and processed one-by-one in a single *event loop,* which normally runs as a separate thread. As discussed in § 4.1, this design can be extended to support additional concurrency by (among other tactics) creating multiple event loop threads, each concurrently processing events, or even dispatching each event in a separate thread. Again, this opens up new design possibilities, but also introduces new concerns about interference and coordination among concurrent activities.

1.2.3.3 Concurrent systems programming

Object-oriented concurrent programming differs from multithreaded systems programming in languages such as C mainly due to the encapsulation, modularity, extensibility, security, and safety features otherwise lacking in C. Additionally, concurrency support is built into the Java programming language, rather than supplied by libraries. This eliminates the possibility of some common errors, and also enables compilers to automatically and safely perform some optimizations that would need to be performed manually in C.

While concurrency support constructs in the Java programming language are generally similar to those in the standard POSIX pthreads library and related packages typically used in C, there are some important differences, especially in the details of waiting and notification (see § 3.2.2). It is very possible to use utility classes that act almost just like POSIX routines (see § 3.4.4). But it is often more productive instead to make minor adjustments to exploit the versions that the language directly supports.

1.2.3.4 Other concurrent programming languages

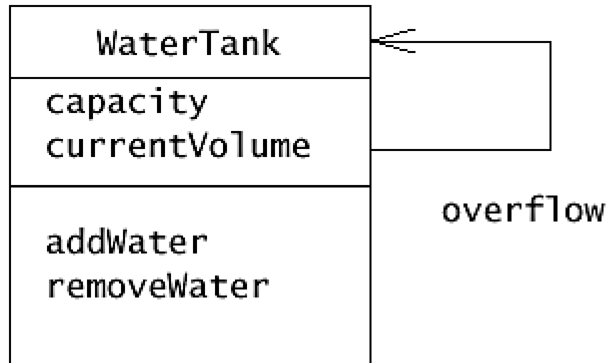
Essentially all concurrent programming languages are, at some level, equivalent, if only in the sense that all concurrent languages are widely believed not to have defined the right concurrency features. However, it is not all that hard to make programs in one language *look* almost equivalent to those in other languages or those using other constructs, by developing packages, classes, utilities, tools, and coding conventions that mimic features built into others. In the course of this book, constructions are introduced that provide the capabilities and programming styles of semaphore-based systems (§ 3.4.1), futures (§ 4.3.3), barrier-based parallelism (§ 4.4.3), CSP (§ 4.5.1) and others. It is a perfectly great idea to write programs using *only* one of these styles, if this suits your needs. However, many concurrent designs, patterns, frameworks, and systems have eclectic heritages and steal good ideas from anywhere they can.

1.2.4 Object Models and Mappings

Conceptions of objects often differ across sequential versus concurrent OO programming, and even across different styles of concurrent OO programming. Contemplation of the underlying object models and mappings can reveal the nature of differences among programming styles hinted at in the previous section.

Most people like to think of software objects as models of real objects, represented with some arbitrary degree of precision. The notion of "real" is of course in the eye of the beholder, and often includes artifices that make sense only within the realm of computation.

For a simple example, consider the skeletal UML class diagram and code sketch for class `WaterTank`:



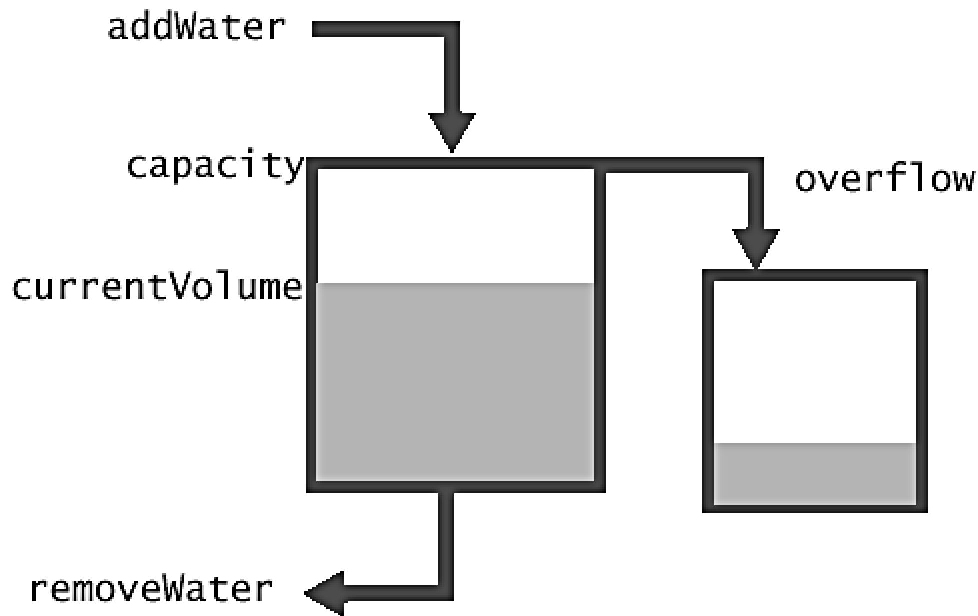

```
class WaterTank {                          // Code sketch
 final float capacity;
 float currentVolume = 0.0f;
 WaterTank overflow;

 WaterTank(float cap) { capacity = cap; ... }

 void addWater(float amount) throws OverflowException;
 void removeWater(float amount) throws UnderflowException;
}
```

The intent here is to represent, or simulate, a water tank with:

- *Attributes* such as `capacity` and `currentVolume`, that are represented as *fields* of `WaterTank` objects. We can choose only those attributes that we happen to care about in some set of usage contexts. For example, while all real water tanks have locations, shapes, colors, and so on, this class only deals with volumes.



- *Invariant* state constraints, such as the facts that the `currentVolume` always remains between zero and `capacity`, and that `capacity` is nonnegative and never changes after construction.
- *Operations* describing behaviors such as those to `addWater` and `removeWater`. This choice of operations again reflects some implicit design decisions concerning accuracy, granularity and precision. For example, we could have chosen to model water tanks at the level of valves and switches, and could have modeled each water molecule as an object that changes location as the result of the associated operations.
- *Connections* (and *potential* connections) to other objects with which objects communicate, such as pipes or other tanks. For example, excess water encountered in an `addWater` operation could be shunted to an overflow tank that is known by each tank.
- *Preconditions and postconditions* on the effects of operations, such as rules stating that it is impossible to remove water from an empty tank, or to add water to a full tank that is not equipped with an available overflow tank.
- *Protocols* constraining when and how messages (operation requests) are processed. For example, we may impose a rule that at most one `addWater` or `removeWater` message is processed at any given time or, alternatively, a rule stating that `removeWater` messages are allowed in the midst of `addWater` operations.

1.2.4.1 Object models

The `WaterTank` class uses objects to model reality. Object models provide rules and frameworks for defining objects more generally, covering:

Statics. The structure of each object is described (normally via a class) in terms of internal attributes (state), connections to other objects, local (internal) methods, and methods or ports for accepting messages from other objects.

Encapsulation. Objects have membranes separating their insides and outsides. Internal state can be directly modified only by the object itself. (We ignore for now language features that allow this rule to be broken.)

Communication. Objects communicate only via message passing. Objects issue messages that trigger actions in other objects. The forms of these messages may range from simple procedural calls to those transported via arbitrary communication protocols.

Identity. New objects can be constructed at any time (subject to system resource constraints) by any object (subject to access control). Once constructed, each object maintains a unique identity that persists over its lifetime.

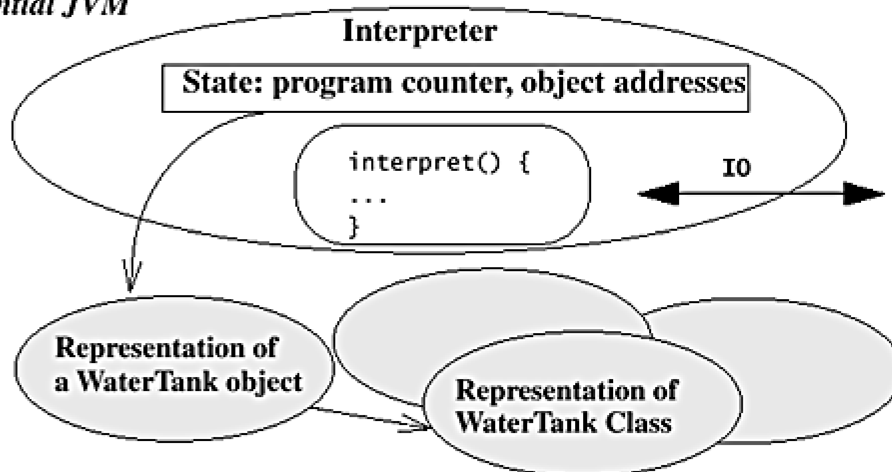
Connections. One object can send messages to others if it knows their identities. Some models rely on *channel* identities rather than or in addition to object identities. Abstractly, a channel is a vehicle for passing messages. Two objects that share a channel may pass messages through that channel without knowing each other's identities. Typical OO models and languages rely on object-based primitives for direct method invocations, channel-based abstractions for IO and communication across wires, and constructions such as event channels that may be viewed from either perspective.

Computation. Objects may perform four basic kinds of computation:

- Accept a message.
- Update internal state.
- Send a message.
- Create a new object.

This abstract characterization can be interpreted and refined in several ways. For example, one way to implement a `WaterTank` object is to build a tiny special-purpose hardware device that only maintains the indicated states, instructions, and connections. But since this is not a book on hardware design, we'll ignore such options and restrict attention to software-based alternatives.

1.2.4.2 Sequential mappings



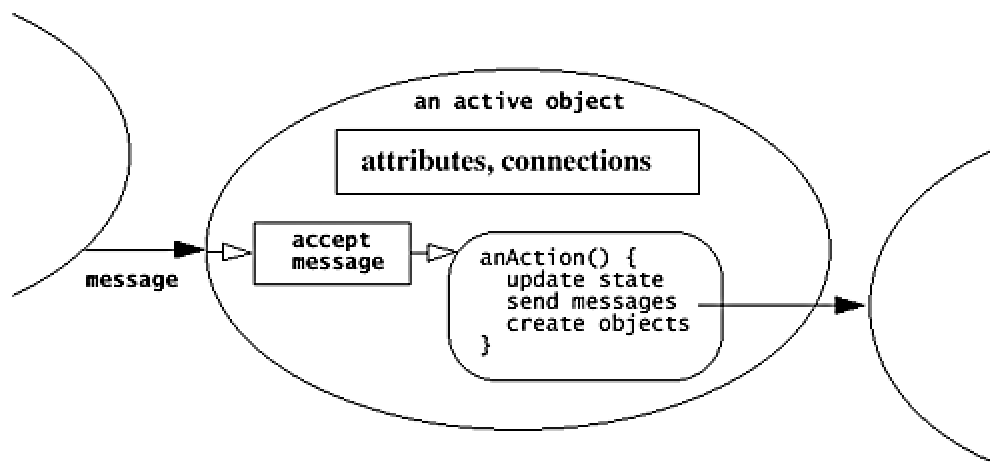
The features of an ordinary general-purpose computer (a CPU, a bus, some memory, and some IO ports) can be exploited so that this computer can pretend it is any object, for example a `WaterTank`. This can be arranged by loading a description of `WaterTanks` (via a `.class` file) into a JVM. The JVM can then construct a *passive* representation of an instance and then interpret the associated operations. This mapping strategy also applies at the level of the CPU when operations are compiled into native code rather than interpreted as bytecodes. It also extends to

programs involving many objects of different classes, each loaded and instantiated as needed, by having the JVM at all times record the identity ("`this`") of the object it is currently simulating.

In other words, the JVM is itself an object, although a very special one that can pretend it is any other object. (More formally, it serves as a Universal Turing Machine.) While similar remarks hold for the mappings used in most other languages, `Class` objects and reflection make it simpler to characterize reflective objects that treat other objects as data.

In a purely sequential environment, this is the end of the story. But before moving on, consider the restrictions on the generic object model imposed by this mapping. On a sequential JVM, it would be impossible to directly simulate multiple concurrent interacting `waterTank` objects. And because all message-passing is performed via sequential procedural invocation, there is no need for rules about whether multiple messages may be processed concurrently — they never are anyway. Thus, sequential OO processing limits the kinds of high-level design concerns you are allowed to express.

1.2.4.3 Active objects

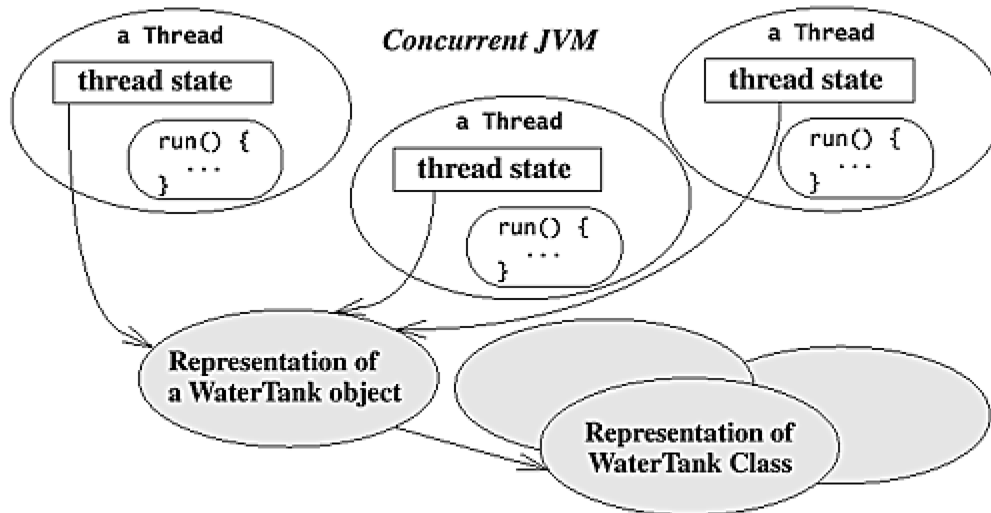


At the other end of the mapping spectrum are *active* object models (also known as *actor* models), in which *every* object is autonomous. Each may be as powerful as a sequential JVM. Internal class and object representations may take the same forms as those used in passive frameworks. For example here, each `waterTank` could be mapped to a separate active object by loading in a description to a separate JVM, and then forever allowing it to simulate the defined actions.

Active object models form a common high-level view of objects in distributed object-oriented systems: Different objects may reside on different machines, so the location and administrative domain of an object are often important programming issues. All message passing is arranged via remote communication (for example via sockets) that may obey any of a number of protocols, including oneway messaging (i.e., messages that do not intrinsically require replies), multicasts (simultaneously sending the same message to multiple recipients), and procedure-style request-reply exchanges.

This model also serves as an object-oriented view of most operating-system-level *processes,* each of which is as independent of, and shares as few resources with, other processes as possible (see § 1.2.2).

1.2.4.4 Mixed models

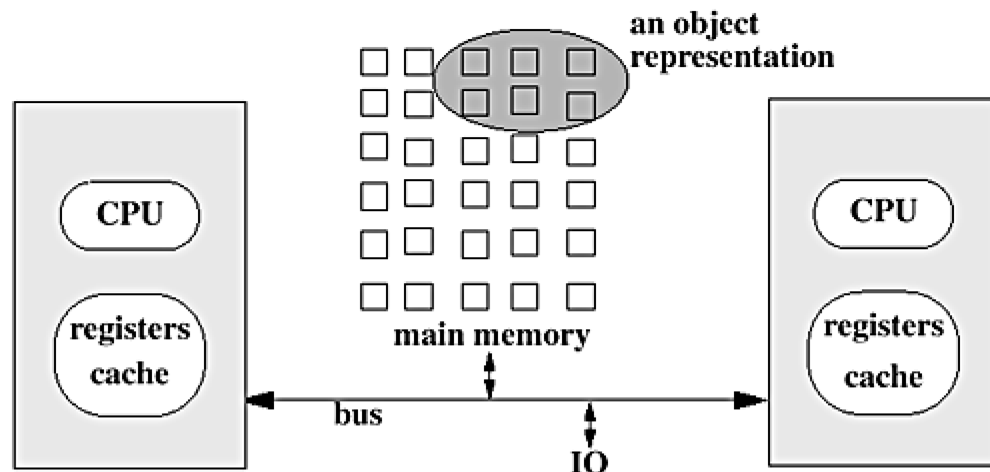


The models and mappings underlying concurrency support in the Java programming language fall between the two extremes of passive and active models. A full JVM may be composed of multiple threads, each of which acts in about the same way as a single sequential JVM. However, unlike pure active objects, all of these threads may share access to the same set of underlying passive representations.

This style of mapping can simulate each of the extremes. Purely passive sequential models can be programmed using only one thread. Purely active models can be programmed by creating as many threads as there are active objects, avoiding situations in which more than one thread can access a given passive representation (see § 2.3), and using constructs that provide the same semantic effects as remote message passing (see § 4.1). However, most concurrent programs occupy a middle ground.

Thread-based concurrent OO models conceptually separate "normal" passive objects from active objects (threads). But the passive objects typically display thread-awareness not seen in sequential programming, for example by protecting themselves via locks. And the active objects are simpler than those seen in actor models, supporting only a few operations (such as `run`). But the design of concurrent OO systems can be approached from either of these two directions — by smartening up passive objects to live in a multithreaded environment, or by dumbing down active objects so they can be expressed more easily using thread constructs.

One reason for supporting this kind of object model is that it maps in a straightforward and efficient way to stock uniprocessor and shared-memory multiprocessor (SMP) hardware and operating systems: Threads can be bound to CPUs when possible and desirable, and otherwise time-shared; local thread state maps to registers and CPUs; and shared object representations map to shared main memory.



The degree of programmer control over these mappings is one distinction separating many forms of *parallel* programming from concurrent programming. Classic parallel programming involves explicit design steps to map threads, tasks, or processes, as well as data, to physical processors and their local stores. Concurrent programming leaves most mapping decisions to the JVM (and the underlying OS). This enhances portability, at the expense of needing to accommodate differences in the quality of implementation of these mappings.

Time-sharing is accomplished by applying the same kind of mapping strategy to threads themselves: Representations of `Thread` objects are maintained, and a scheduler arranges *context switches* in which the CPU state corresponding to one thread is saved in its associated storage representation and restored from another.

Several further refinements and extensions of such models and mappings are possible. For example, *persistent* object applications and systems typically rely on databases to maintain object representations rather than directly relying on main memory.

1.2.5 Further Readings

There is a substantial literature on concurrency, ranging from works on theoretical foundations to practical guides for using particular concurrent applications.

1.2.5.1 Concurrent programming

Textbooks presenting details on additional concurrent algorithms, programming strategies, and formal methods not covered in this book include:

Andrews, Gregory. *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 1999. This is an expanded update of Andrews's *Concurrent Programming: Principles and Practice*, Benjamin Cummings, 1991.

Ben-Ari, M. *Principles of Concurrent and Distributed Programming*, Prentice Hall, 1990.

Bernstein, Arthur, and Philip Lewis. *Concurrency in Programming and Database Systems*, Jones and Bartlett, 1993.

Burns, Alan, and Geoff Davis. *Concurrent Programming*, Addison-Wesley, 1993.

Bustard, David, John Elder, and Jim Welsh. *Concurrent Program Structures*, Prentice Hall, 1988.

Schneider, Fred. *On Concurrent Programming*, Springer-Verlag, 1997.

The concurrency constructs found in the Java programming language have their roots in similar constructs first described by C. A. R. Hoare and Per Brinch Hansen. See papers by them and others in following collections:

Dahl, Ole-Johan, Edsger Dijkstra, and C. A. R. Hoare (eds.). *Structured Programming*, Academic Press, 1972.

Gehani, Narain, and Andrew McGettrick (eds.). *Concurrent Programming*, Addison-Wesley, 1988.

A comparative survey of how some of these constructs are defined and supported across different languages and systems may be found in:

Buhr, Peter, Michel Fortier, and Michael Coffin. "Monitor Classification", *ACM Computing Surveys*, 1995.

Concurrent object-oriented, object-based or module-based languages include Simula, Modula-3, Mesa, Ada, Orca, Sather, and Euclid. More information on these languages can be found in their manuals, as well as in:

Birtwistle, Graham, Ole-Johan Dahl, Bjorn Myhrtag, and Kristen Nygaard. *Simula Begin*, Auerbach Press, 1973.

Burns, Alan, and Andrew Wellings. *Concurrency in Ada*, Cambridge University Press, 1995.

Holt, R. C. *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, 1983.

Nelson, Greg (ed.). *Systems Programming with Modula-3*, Prentice Hall, 1991.

Stoutamire, David, and Stephen Omohundro. *The Sather/pSather 1.1 Specification*, Technical Report, University of California at Berkeley, 1996.

Books taking different approaches to concurrency in the Java programming language include:

Hartley, Stephen. *Concurrent Programming using Java*, Oxford University Press, 1998. This takes an operating systems approach to concurrency.

Holub, Allen. *Taming Java Threads*, Apress, 1999. This collects the author's columns on threads in the *JavaWorld* online magazine.

Lewis, Bil. *Multithreaded Programming in Java*, Prentice Hall, 1999. This presents a somewhat lighter treatment of several topics discussed in this book, and provides closer tie-ins with POSIX threads.

Magee, Jeff, and Jeff Kramer. *Concurrency: State Models and Java Programs*, Wiley, 1999. This provides a stronger emphasis on modeling and analysis.

Most books, articles, and manuals on systems programming using threads concentrate on the details of those on particular operating systems or thread packages. See:

Butenhof, David. *Programming with POSIX Threads*, Addison-Wesley, 1997. This provides the most complete discussions of the POSIX thread library and how to use it.

Lewis, Bil, and Daniel Berg. *Multithreaded Programming with Pthreads*, Prentice Hall, 1998.

Norton, Scott, and Mark Dipasquale. *Thread Time,* Prentice Hall, 1997.

Most texts on operating systems and systems programming describe the design and construction of underlying support mechanisms for language-level thread and synchronization constructs. See, for example:

Hanson, David. *C Interfaces and Implementations*, Addison-Wesley, 1996.

Silberschatz, Avi and Peter Galvin. *Operating Systems Concepts*, Addison-Wesley, 1994.

Tanenbaum, Andrew. *Modern Operating Systems*, Prentice Hall, 1992.

1.2.5.2 Models

Given the diverse forms of concurrency seen in software, it's not surprising that there have been a large number of approaches to the basic theory of concurrency. Theoretical accounts of process calculi, event structures, linear logic, Petri nets, and temporal logic have potential relevance to the understanding of concurrent OO systems. For overviews of most approaches to the theory of concurrency, see:

van Leeuwen, Jan (ed.). *Handbook of Theoretical Computer Science, Volume B*, MIT Press, 1990.

An eclectic (and still fresh-sounding) presentation of models, associated programming techniques, and design patterns, illustrated using diverse languages and systems, is:

Filman, Robert, and Daniel Friedman. *Coordinated Computing*. McGraw-Hill, 1984.

There are several experimental concurrent OO languages based on active objects, most notably the family of *Actor* languages. See:

Agha, Gul. *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

A more extensive survey of object-oriented approaches to concurrency can be found in:

Briot, Jean-Pierre, Rachid Guerraoui, and Klaus-Peter Lohr. "Concurrency and Distribution in Object-Oriented Programming", *Computing Surveys*, 1998.

Research papers on object-oriented models, systems and languages can be found in proceedings of OO conferences including *ECOOP, OOPSLA, COOTS, TOOLS,* and *ISCOPE,* as well as concurrency

conferences such as *CONCUR* and journals such as *IEEE Concurrency.* Also, the following collections contain chapters surveying many approaches and issues:

Agha, Gul, Peter Wegner, and Aki Yonezawa (eds.). *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

Briot, Jean-Pierre, Jean-Marc Geib and Akinori Yonezawa (eds.). *Object Based Parallel and Distributed Computing*, LNCS 1107, Springer Verlag, 1996.

Guerraoui, Rachid, Oscar Nierstrasz, and Michel Riveill (eds.). *Object-Based Distributed Processing*, LNCS 791, Springer-Verlag, 1993.

Nierstrasz, Oscar, and Dennis Tsichritzis (eds.). *Object-Oriented Software Composition*, Prentice Hall, 1995.

1.2.5.3 Distributed systems

Texts on distributed algorithms, protocols, and system design include:

Barbosa, Valmir. *An Introduction to Distributed Algorithms*. Morgan Kaufman, 1996.

Birman, Kenneth and Robbert von Renesse. *Reliable Distributed Computing with the Isis Toolkit*, IEEE Press, 1994.

Coulouris, George, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*, Addison-Wesley, 1994.

Lynch, Nancy. *Distributed Algorithms*, Morgan Kaufman, 1996.

Mullender, Sape (ed.), *Distributed Systems*, Addison-Wesley, 1993.

Raynal, Michel. *Distributed Algorithms and Protocols*, Wiley, 1988.

For details about distributed programming using RMI, see:

Arnold, Ken, Bryan O'Sullivan, Robert Scheifler, Jim Waldo, and Ann Wollrath. *The Jini™ Specification*, Addison-Wesley, 1999.

1.2.5.4 Real-time programming

Most texts on real-time programming focus on *hard real-time* systems in which, for the sake of correctness, certain activities must be performed within certain time constraints. The Java programming language does not supply primitives that provide such guarantees, so this book does not cover deadline scheduling, priority assignment algorithms, and related concerns. Sources on real-time design include:

Burns, Alan, and Andy Wellings. *Real-Time Systems and Programming Languages*, Addison-Wesley, 1997. This book illustrates real-time programming in Ada, occam, and C, and includes a recommended account of priority inversion problems and solutions.

Gomaa, Hassan. *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, 1993.

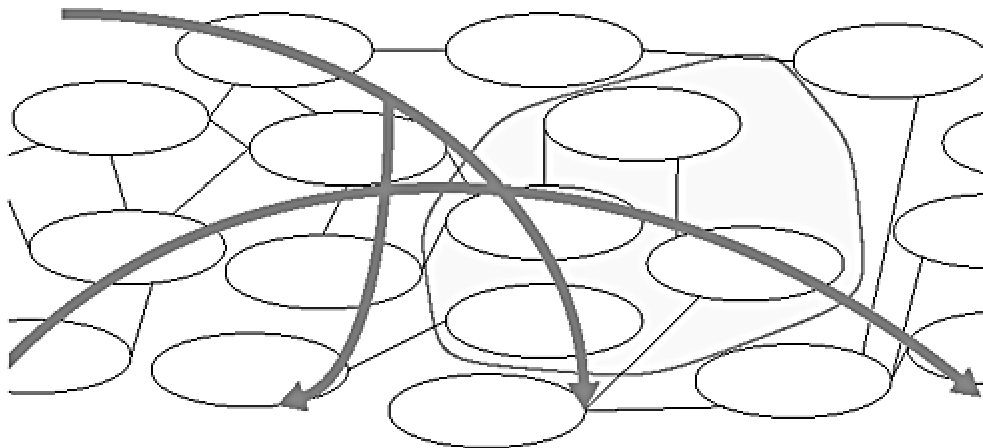
Levi, Shem-Tov and Ashok Agrawala. *Real-Time System Design*, McGraw-Hill, 1990.

Selic, Bran, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*, Wiley, 1995.

1.3 Design Forces

This section surveys design concerns that arise in concurrent software development, but play at best minor roles in sequential programming. Most presentations of constructions and design patterns later in this book include descriptions of how they resolve applicable forces discussed here (as well as others that are less directly tied to concurrency, such as accuracy, testability, and so on).

One can take two complementary views of any OO system, object-centric and activity-centric:



Under an object-centric view, a system is a collection of interconnected objects. But it is a structured collection, not a random object soup. Objects cluster together in groups, for example the group of objects comprising a `ParticleApplet`, thus forming larger components and subsystems.

Under an activity-centric view, a system is a collection of possibly concurrent activities. At the most fine-grained level, these are just individual message sends (normally, method invocations). They in turn organize themselves into sets of call-chains, event sequences, tasks, sessions, transactions, and threads. One logical activity (such as running the `ParticleApplet`) may involve many threads. At a higher level, some of these activities represent system-wide use cases.

Neither view alone provides a complete picture of a system, since a given object may be involved in multiple activities, and conversely a given activity may span multiple objects. However, these two views give rise to two complementary sets of *correctness* concerns, one object-centric and the other activity-centric:

Safety. Nothing bad ever happens to an object.

Liveness. Something eventually happens within an activity.

Safety failures lead to unintended behavior at run time — things just start going wrong. Liveness failures lead to no behavior — things just stop running. Sadly enough, some of the easiest things you can do to improve liveness properties can destroy safety properties, and vice versa. Getting them both right can be a challenge.

You have to balance the relative effects of different kinds of failure in your own programs. But it is a standard engineering (not just software engineering) practice to place primary design emphasis on safety. The more your code actually matters, the better it is to ensure that a program does nothing at all rather than something that leads to random, even dangerous behavior.

On the other hand, most of the time spent tuning concurrent designs in practice usually surrounds liveness and liveness-related efficiency issues. And there are sometimes good, conscientious reasons for selectively sacrificing safety for liveness. For example, it may be acceptable for visual displays to transiently show utter nonsense due to uncoordinated concurrent execution— drawing stray pixels, incorrect progress indicators, or images that bear no relation to their intended forms — if you are confident that this state of affairs will soon be corrected.

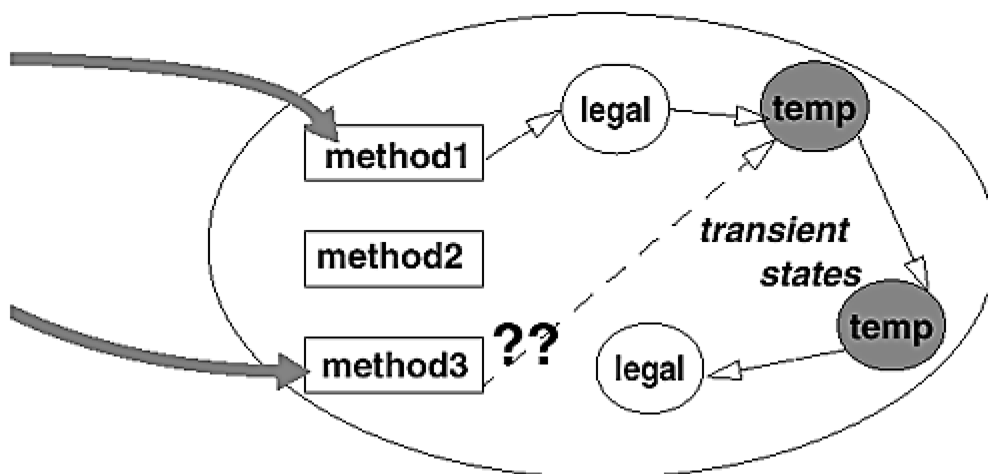
Safety and liveness issues may be further extended to encompass two categories of *quality* concerns, one mainly object-centric and the other mainly activity-centric, that are also sometimes in direct opposition:

Reusability. The utility of objects and classes across multiple contexts.

Performance. The extent to which activities execute soon and quickly.

The remainder of this section looks more closely at safety, liveness, performance, and reusability in concurrent programs. It presents basic terms and definitions, along with brief introductions to core issues and tactics that are revisited and amplified throughout the course of this book.

1.3.1 Safety



Safe concurrent programming practices are generalizations of safe and secure sequential programming practices. Safety in concurrent designs adds a temporal dimension to common notions of *type* safety.

A type-checked program might not be correct, but at least it doesn't do dangerous things like misinterpret the bits representing a `float` as if they were an object reference. Similarly, a safe concurrent design might not have the intended effect, but at least it never encounters errors due to corruption of representations by contending threads.

One practical difference between type safety and multithreaded safety is that most type-safety matters can be checked automatically by compilers. A program that fails to pass compile-time checks cannot even be run. Most multithreaded safety matters, however, cannot be checked automatically, and so must rely on programmer discipline. Methods for *proving* designs to be safe fall outside the scope of this book (see the Further Readings). The techniques for ensuring safety described here rely on careful engineering practices (including several with roots in formalisms) rather than formal methods themselves.

Multithreaded safety also adds a temporal dimension to design and programming techniques surrounding *security.* Secure programming practices disable access to certain operations on objects and resources from certain callers, applications, or principals. Concurrency control introduces *transient* disabling of access based on consideration of the actions currently being performed by other threads.

The main goal in safety preservation is ensuring that all objects in a system maintain *consistent* states: states in which all fields, and all fields of other objects on which they depend, possess legal, meaningful values. It sometimes takes hard work to nail down exactly what "legal" and "meaningful" mean in a particular class. One path is first to establish conceptual-level *invariants,* for example the rule that water tank volumes must always be between zero and their capacities. These can usually be recast in terms of relationships among field values in the associated concrete classes.

An object is *consistent* if all fields obey their invariants. Every public method in every class should lead an object from one consistent state to another. Safe objects may occasionally enter transiently inconsistent states in the midst of methods, but they never attempt to initiate new actions when they are in inconsistent states. If every object is designed to perform actions only when it is logically able to do so, and if all the mechanics are properly implemented, then you can be sure that an application using these objects will not encounter any errors due to object inconsistency.

One reason for being more careful about invariants in concurrent programs is that it is much easier to break them inadvertently than in most sequential programs. The need for protection against the effects of inconsistency arises even in sequential contexts, for example when processing exceptions and callbacks, and when making self-calls from one method in a class to another. However, these issues become much more central in concurrent programs. As discussed in § 2.2, the most common ways of ensuring consistency employ exclusion techniques to guarantee the *atomicity* of public actions — that each action runs to completion without interference from others. Without such protection, inconsistencies in concurrent programs may stem from *race conditions* producing *storage conflicts* at the level of raw memory cells:

Read/Write conflicts. One thread reads a value of a field while another writes to it. The value seen by the reading thread is difficult to predict — it depends on which thread won the "race" to access the field first. As discussed in § 2.2, the value read need not even be a value that was ever written by any thread.

Write/Write conflicts. Two threads both try to write to the same field. The value seen upon the next read is again difficult or impossible to predict.

It is equally impossible to predict the consequences of actions that are attempted when objects are in inconsistent states. Examples include:

- A graphical representation (for example of a `Particle`) is displayed at a location that the object never actually occupied.
- A bank account balance is incorrect after an attempt to withdraw money in the midst of an automatic transfer.
- Following the `next` pointer of a linked list leads to a node that is not even in the list.
- Two concurrent sensor updates cause a real-time controller to perform an incorrect effector action.

1.3.1.1 Attributes and constraints

Safe programming techniques rely on clear understanding of required properties and constraints surrounding object representations. Developers who are not aware of these properties rarely do a very good job at preserving them. Many formalisms are available for precisely stating predicates describing requirements (as discussed in most of the texts on concurrent design methods listed in the Further Readings). These can be very useful, but here we will maintain sufficient precision without introducing formalisms.

Consistency requirements sometimes stem from definitions of high-level conceptual attributes made during the initial design of classes. These constraints typically hold regardless of how the attributes are concretely represented and accessed via fields and methods. This was seen for example in the development of the `WaterTank` and `Particle` classes earlier in this chapter. Here are some other examples, most of which are revisited in more detail in the course of this book:

- A `BankAccount` has a *balance* that is equal to the sum of all deposits and interest minus withdrawals and service charges.
- A `Packet` has a *destination* that must be a legal IP address.
- A `Counter` has a nonnegative integral *count* value.
- An `Invoice` has a *paymentDue* that reflects the rules of a payment system.
- A `Thermostat` has a *temperature* equal to the most recent sensor reading.
- A `Shape` has a *location, dimension,* and *color* that all obey a set of stylistic guidelines for a given GUI toolkit.
- A `BoundedBuffer` has an *elementCount* that is always between zero and a *capacity.*
- A `Stack` has a *size* and, when not empty, a *top* element.
- A `Window` has a *propertySet* maintaining current mappings of fonts, background color, etc.
- An `Interval` has a *startDate* that is no later than its *endDate.*

While such attributes essentially always somehow map to object fields, the correspondences need not be direct. For example, the `top` of a `Stack` is typically not held in a variable, but instead in an array element or linked list node. Also, some attributes can be computed ("derived") via others; for example, the boolean attribute `overdrawn` of a `BankAccount` might be computed by comparing the balance to zero.

1.3.1.2 Representational constraints

Further constraints and invariants typically emerge as additional implementation decisions are made for a given class. Fields declared for the sake of maintaining a particular data structure, for improving

performance, or for other internal bookkeeping purposes often need to respect sets of invariants. Broad categories of fields and constraints include the following:

Direct value representations. Fields needed to implement concrete attributes. For example, a `Buffer` might have a `putIndex` field holding the array index position to use when inserting the next added element.

Cached value representations. Fields used to eliminate or minimize the need for computations or method invocations. For example, rather than computing the value of `overdrawn` every time it is needed, a `BankAccount` might maintain an `overdrawn` field that is true if and only if the current balance is less than zero.

Logical state representations. Reflections of logical control state. For example, a `BankCardReader` might have a `card` field representing the card currently being read, and a `validPIN` field recording whether the PIN access code was verified. The `CardReader` `validPIN` field may be used to track the point in a protocol in which the card has been successfully read in and validated. Some state representations take the form of *role variables,* controlling responses to all of a related set of methods (sometimes those declared in a single interface). For example, a game-playing object may alternate between active and passive roles depending on the value of a `whoseTurn` field.

Execution state variables. Fields recording the fine-grained dynamic state of an object, for example, the fact that a certain operation is in progress. Execution state variables can represent the fact that a given message has been received, that the corresponding action has been initiated, that the action has terminated, and that a reply to the message has been issued. An execution state variable is often an enumerated type with values having names ending in *-ing;* for example, `CONNECTING`, `UPDATING`, `WAITING`. Another common kind of execution state variable is a counter that records the number of entries or exits of some method. As discussed in § 3.2, objects in concurrent programs tend to require more such variables than do those in sequential contexts, to help track and manage the progress of methods that proceed asynchronously.

History variables. Representations of the history or past states of an object. The most extensive representation is a *history log,* recording all messages ever received and sent, along with all corresponding internal actions and state changes that have been initiated and completed. Less extensive subsets are much more common. For example, a `BankAccount` class could maintain a `lastSavedBalance` field that holds the last checkpointed value and is used when reverting cancelled transactions.

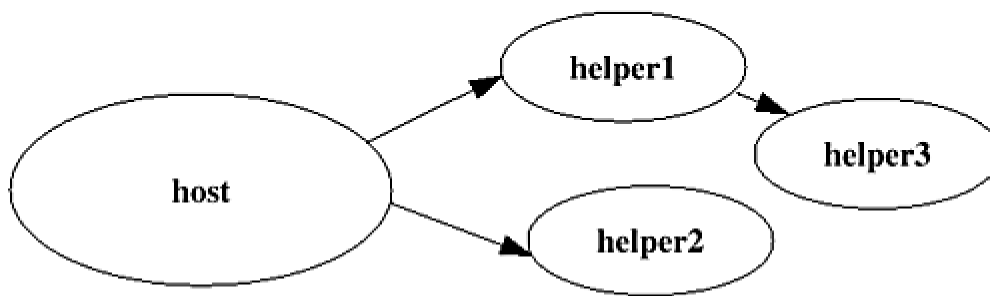
Version tracking variables. An integer, time-stamp, object reference, signature code, or other representation indicating the time, ordering, or nature of the last state change made by an object. For example, a `Thermostat` may increment a `readingNumber` or record the `lastReadingTime` when updating its `temperature`.

References to acquaintances. Fields pointing to other objects that the host interacts with, but that do not themselves comprise the host's logical state: For example, a `callback` target of an `EventDispatcher`, or a `requestHandler` delegated to by a `WebServer`.

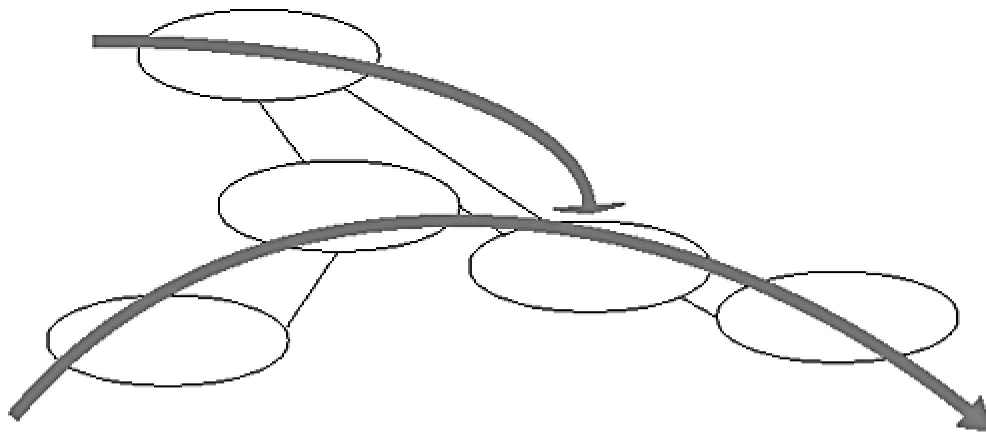
References to representation objects. Attributes that are conceptually held by a host object but are actually managed by other helper objects. Reference fields may point to other objects that assist in representing the state of the host object. So, the logical state of any object may include the states of

objects that it holds references to. Additionally, the reference fields themselves form part of the concrete state of the host object (see § 2.3.3). Any attempts to ensure safety must take these relationships into account. For example:

- A `Stack` might have a `headOfLinkedList` field recording the first node of a list representing the stack.
- A `Person` object might maintain a `homePageURL` field maintained as a `java.net.URL` object.
- The balance of a `BankAccount` might be maintained in a central repository, in which case the `BankAccount` would instead maintain a a field referring to the repository (in order to ask it about the current balance). In this case, some of the logical state of the `BankAccount` is actually managed by the repository.
- An object might know of its attributes only via access to property lists maintained by other objects.



1.3.2 Liveness



One way to build a guaranteed safe system is to arrange that no objects ever execute any methods, and thus can never encounter any conflicts. But this is not a very productive form of programming. Safety concerns must be balanced by liveness[1] concerns.

[1] Some "liveness" properties may be construed as safety properties of sets of thread objects. For example, deadlock-freedom may be defined as avoiding the bad state in which a set of threads endlessly wait for each other.

In live systems, every activity eventually progresses toward completion; every invoked method eventually executes. But an activity may (perhaps only transiently) fail to make progress for any of several interrelated reasons:

Locking. A `synchronized` method blocks one thread because another thread holds the lock.

Waiting. A method blocks (via `Object.wait` or its derivatives) waiting for an event, message, or condition that has yet to be produced within another thread.

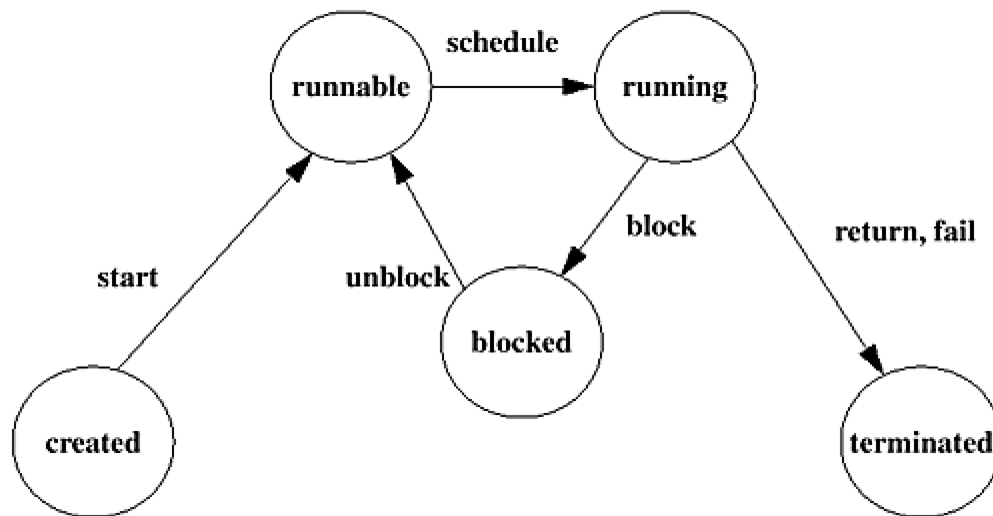
Input. An IO-based method waits for input that has not yet arrived from another process or device.

CPU contention. A thread fails to run even though it is in a runnable state because other threads, or even completely separate programs running on the same computer, are occupying CPU or other computational resources.

Failure. A method running in a thread encounters a premature exception, error, or fault.

Momentary blockages in thread progress are usually acceptable. In fact, frequent short-lived blocking is intrinsic to many styles of concurrent programming.

The lifecycle of a typical thread may include a number of transient blockages and reschedulings:



However, *permanent* or unbounded lack of progress is usually a serious problem. Examples of potentially permanent liveness failures described in more depth elsewhere in this book include:

Deadlock. Circular dependencies among locks. In the most common case, thread `A` holds a lock for object `X` and then tries to acquire the lock for object `Y`. Simultaneously, thread `B` already holds the lock for object `Y` and tries to acquire the lock for object `X`. Neither thread can ever make further progress (see § 2.2.5).

Missed signals. A thread remains dormant because it started waiting *after* a notification to wake it up was produced (see § 3.2.2).

Nested monitor lockouts. A waiting thread holds a lock that would be needed by any other thread attempting to wake it up (see § 3.3.4).

Livelock. A continuously retried action continuously fails (see § 2.4.4.2).

Starvation. The JVM/OS fails ever to allocate CPU time to a thread. This may be due to scheduling policies or even hostile denial-of-service attacks on the host computer (see § 1.1.2.3 and § 3.4.1.5).

Resource exhaustion. A group of threads together hold all of a finite number of resources. One of them needs additional resources, but no other thread will give one up (see § 4.5.1).

Distributed failure. A remote machine connected by a socket serving as an `InputStream` crashes or becomes inaccessible (see § 3.1).

1.3.3 Performance

Performance-based forces extend liveness concerns. In addition to demanding that every invoked method eventually execute, performance goals require them to execute soon and quickly. While we do not consider in this book hard real-time systems in which failure to execute within a given time interval can lead to catastrophic system errors, nearly all concurrent programs have implicit or explicit performance goals.

Meaningful performance requirements are stated in terms of measurable qualities, including the following metrics. Goals may be expressed for central tendencies (e.g., mean, median) of measurements, as well as their variability (e.g., range, standard deviation).

Throughput. The number of operations performed per unit time. The operations of interest may range from individual methods to entire program runs. Most often, throughput is reported not as a rate, but instead as the time taken to perform one operation.

Latency. The time elapsed between issuing a message (via for example a mouse click, method invocation, or incoming socket connection) and servicing it. In contexts where operations are uniform, single-threaded, and "continuously" requested, latency is just the inverse of throughput. But more typically, the latencies of interest reflect response times — the delays until *something* happens, not necessarily full completion of a method or service.

Capacity. The number of simultaneous activities that can be supported for a given target minimum throughput or maximum latency. Especially in networking applications, this can serve as a useful indicator of overall *availability,* since it reflects the number of clients that can be serviced without dropping connections due to time-outs or network queue overflows.

Efficiency. Throughput divided by the amount of computational resources (for example CPUs, memory, and IO devices) needed to obtain this throughput.

Scalability. The rate at which latency or throughput improves when resources (again, usually CPUs, memory, or devices) are added to a system. Related measures include *utilization* — the percentage of available resources that are applied to a task of interest.

Degradation. The rate at which latency or throughput worsens as more clients, activities, or operations are added without adding resources.

Most multithreaded designs implicitly accept a small trade-off of poorer computational efficiency to obtain better latency and scalability. Concurrency support introduces the following kinds of overhead and contention that can slow down programs:

Locks. A `synchronized` method typically requires greater call overhead than an unsynchronized method. Also, methods that frequently block waiting for locks (or for any other reason) proceed more slowly than those that do not.

Monitors. `Object.wait`, `Object.notify`, `Object.notifyAll`, and the methods derived from them (such as `Thread.join`) can be more expensive than other basic JVM run-time support operations.

Threads. Creating and starting a `Thread` is typically more expensive than creating an ordinary object and invoking a method on it.

Context-switching. The mapping of threads to CPUs encounters context-switch overhead when a JVM/OS saves the CPU state associated with one thread, selects another thread to run, and loads the associated CPU state.

Scheduling. Computations and underlying policies that select which eligible thread to run add overhead. These may further interact with other system chores such as processing asynchronous events and garbage collection.

Locality. On multiprocessors, when multiple threads running on different CPUs share access to the same objects, cache consistency hardware and low-level system software must communicate the associated values across processors.

Algorithmics. Some efficient sequential algorithms do not apply in concurrent settings. For example, some data structures that rely on caching work only if it is known that exactly one thread performs all operations. However, there are also efficient alternative concurrent algorithms for many problems, including those that open up the possibility of further speedups via parallelism.

The overheads associated with concurrency constructs steadily decrease as JVMs improve. For example, as of this writing, the overhead cost of a single uncontended `synchronized` method call with a no-op body on recent JVMs is on the order of a few unsynchronized no-op calls. (Since different kinds of calls, for example of static versus instance methods, can take different times and interact with other optimizations, it is not worth making this more precise.)

However, these overheads tend to degrade nonlinearly. For example, using one lock that is frequently contended by ten threads is likely to lead to much poorer overall performance than having each thread pass through ten uncontended locks. Also, because concurrency support entails underlying system resource management that is often optimized for given target loads, performance can dramatically degrade when too many locks, monitor operations, or threads are used.

Subsequent chapters include discussions of minimizing use of the associated constructs when necessary. However, bear in mind that performance problems of any kind can be remedied only after they are measured and isolated. Without empirical evidence, most guesses at the nature and source of performance problems are wrong. The most useful measurements are comparative, showing differences or trends under different designs, loads, or configurations.

1.3.4 Reusability

A class or object is reusable to the extent that it can be readily employed across different contexts, either as a black-box component or as the basis of white-box extension via subclassing and related techniques.

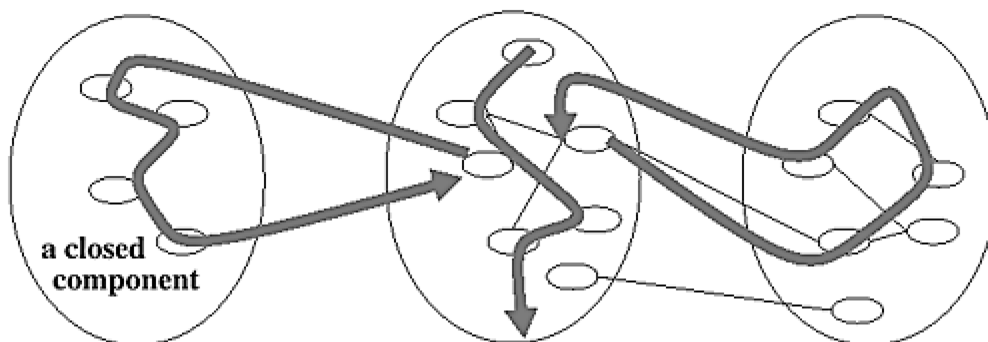
The interplay between safety and liveness concerns can significantly impact reusability. It is usually possible to design components to be safe across *all* possible contexts. For example, a `synchronized` method that refuses to commence until it possesses the synchronization lock will do this no matter how it is used. But in some of these contexts, programs using this safe component might encounter liveness failures (for example, deadlock). Conversely, the functionality surrounding a component using only unsynchronized methods will *always* be live (at least with respect to locking), but may encounter safety violations when multiple concurrent executions are allowed to occur.

The dualities of safety and liveness are reflected in some extreme views of design methodology. Some top-down design strategies take a pure safety-first approach: Ensure that each class and object is safe, and then later try to improve liveness as an optimization measure. An opposite, bottom-up approach is sometimes adopted in multithreaded systems programming: Ensure that code is live, and then try to layer on safety features, for example by adding locks. Neither extreme is especially successful in practice. It is too easy for top-down approaches to result in slow, deadlock-prone systems, and for bottom-up approaches to result in buggy code with unanticipated safety violations.

It is usually more productive to proceed with the understanding that some very useful and efficient components are not, and need not be, absolutely safe, and that useful services supported by some components are not absolutely live. Instead, they operate correctly only within certain restricted usage contexts. Therefore, establishing, documenting, advertising, and exploiting these contexts become central issues in concurrent software design.

There are two general approaches (and a range of intermediate choices) for dealing with context dependence: (1) Minimize uncertainty by closing off parts of systems, and (2) Establish policies and protocols that enable components to become or remain open. Many practical design efforts involve some of each.

1.3.4.1 Closed subsystems



An ideally closed system is one for which you have perfect static (design time) knowledge about all possible behaviors. This is typically both unattainable and undesirable. However, it is often still possible to close off parts of systems, in units ranging from individual classes to product-level components, by employing possibly extreme versions of OO encapsulation techniques:

Restricted external communication. All interactions, both inward and outward, occur through a narrow interface. In the most tractable case, the subsystem is *communication-closed,* never internally invoking methods on objects outside the subsystem.

Deterministic internal structure. The concrete nature (and ideally, number) of all objects and threads comprising the subsystem are statically known. The `final` and `private` keywords can be used to help enforce this.

In at least some such systems, you can in principle prove — informally, formally, or even mechanically — that no *internal* safety or liveness violations are possible within a closed component. Or, if they are possible, you can continue to refine designs and implementations until a component is provably correct. In the best cases, you can then apply this knowledge compositionally to analyze other parts of a system that rely on this component.

Perfect static information about objects, threads and interactions tells you not only what can happen, but also what cannot happen. For example, it may be the case that, even though two `synchronized` methods in two objects contain calls to each other, they can never be accessed simultaneously by different threads within the subsystem, so deadlock will never occur.

Closure may also provide further opportunities for manual or compiler-driven optimization; for example removing synchronization from methods that would ordinarily require it, or employing clever special-purpose algorithms that can be made to apply only by eliminating the possibility of unwanted interaction. Embedded systems are often composed as collections of closed modules, in part to improve predictability, schedulability, and related performance analyses.

While closed subsystems are tractable, they can also be brittle. When the constraints and assumptions governing their internal structure change, these components are often thrown away and redeveloped from scratch.

1.3.4.2 Open systems

An ideal open system is infinitely extensible, across several dimensions. It may load unknown classes dynamically, allow subclasses to override just about any method, employ callbacks across objects within different subsystems, share common resources across threads, use reflection to discover and invoke methods on otherwise unknown objects, and so on. Unbounded openness is usually as unattainable and undesirable as complete closedness: If everything can change, then you cannot program anything. But most systems require at least some of this flexibility.

Full static analysis of open systems is not even possible since their nature and structure evolve across time. Instead, open systems must rely on documented *policies* and *protocols* that every component adheres to.

The Internet is among the best examples of an open system. It continually evolves, for example by adding new hosts, web pages, and services, requiring only that all participants obey a few network policies and protocols. As with other open systems, adherence to Internet policies and protocols is sometimes difficult to enforce. However, JVMs themselves arrange that non-conforming components cannot catastrophically damage system integrity.

Policy-driven design can work well at the much smaller level of typical concurrent systems, where policies and protocols often take the form of design rules. Examples of policy domains explored in more depth in subsequent chapters include:

Flow. For example, a rule of the form: Components of type A send messages to those of type B, but never vice versa.

Blocking. For example, a rule of the form: Methods of type A always immediately throw exceptions if resource R is not available, rather than blocking until it is available.

Notifications. For example, a rule of the form: Objects of type A always send change notifications to their listeners whenever updated.

Adoption of a relatively small number of policies simplifies design by minimizing the possibility of inconsistent case-by-case decisions. Component authors, perhaps with the help of code reviews and tools, need check only that they are obeying the relevant design rules, and can otherwise focus attention on the tasks at hand. Developers can think locally while still acting globally.

However, policy-driven design can become unmanageable when the number of policies grows large and the *programming obligations* they induce overwhelm developers. When even simple methods such as updating an account balance or printing "Hello, world" require dozens of lines of awkward, error-prone code to conform to design policies, it is time to take some kind of remedial action: Simplify or reduce the number of policies; or create tools that help automate code generation and/or check for conformance; or create domain-specific languages that enforce a given discipline; or create frameworks and utility libraries that reduce the need for so much support code to be written inside each method.

Policy choices need not be in any sense "optimal" to be effective, but they must be conformed to and believed in, the more fervently the better. Such policy choices form the basis of several frameworks and design patterns described throughout this book. It is likely that some of them will be inapplicable to your software projects, and may even strike you as wrong-headed ("I'd never do *that*!") because the underlying policies clash with others you have adopted.

While inducing greater closedness allows you to optimize for performance, inducing greater openness allows you to optimize for future change. These two kinds of tunings and refactorings are often equally challenging to carry out, but have opposite effects. Optimizing for performance usually entails exploiting special cases by hard-wiring design decisions. Optimizing for extensibility entails removing hard-wired decisions and instead allowing them to vary, for example by encapsulating them as overridable methods, supporting callback hooks, or abstracting functionality via `interfaces` that can be re-implemented in completely different ways by dynamically loaded components.

Because concurrent programs tend to include more in-the-small policy decisions than sequential ones, and because they tend to rely more heavily on invariants surrounding particular representation choices, classes involving concurrency constructs often turn out to require special attention in order to be readily extensible. This phenomenon is widespread enough to have been given a name, *the inheritance anomaly,* and is described in more detail in § 3.3.3.3.

However, some other programming techniques needlessly restrict extensibility for the sake of performance. These tactics become more questionable as compilers and JVMs improve. For example, dynamic compilation allows many extensible components to be treated as if they are closed at class-loading time, leading to optimizations and specializations that exploit particular run-time contexts more effectively than any programmer could.

1.3.4.3 Documentation

When compositionality is context-dependent, it is vital for intended usage contexts and restrictions surrounding components to be well understood and well documented. When this information is not provided, use, reuse, maintenance, testing, configuration management, system evolution, and related software-engineering concerns are made much more difficult.

Documentation may be used to improve understandability by any of several audiences — other developers using a class as a black-box component, subclass authors, developers who later maintain, modify, or repair code, testers and code reviewers, and system users. Across these audiences, the first goal is to eliminate the need for extensive documentation by minimizing the unexpected, and thus reducing conceptual complexity via:

Standardization. Using common policies, protocols, and interfaces. For example:

- Adopting standard design patterns, and referencing books, web pages, or design documents that describe them more fully.
- Employing standard utility libraries and frameworks.
- Using standard coding idioms and naming conventions.
- Clearing against standard review checklists that enumerate common errors.

Clarity. Using the simplest, most self-evident code expressions. For example:

- Using exceptions to advertise checked conditions.
- Expressing internal restrictions via access qualifiers (such as `private`).
- Adopting common default naming and signature conventions, for example that, unless specified otherwise, methods that can block declare that they throw `InterruptedException`.

Auxiliary code. Supplying code that demonstrates intended usages. For example:

- Including sample or recommended usage examples.
- Providing code snippets that achieve non-obvious effects.
- Including methods designed to serve as self-tests.

After eliminating the need to explain the obvious via documentation, more useful forms of documentation can be used to clarify design *decisions.* The most critical details can be expressed in a systematic fashion, using semiformal annotations of the forms listed in the following table, which are used and further explained as needed throughout this book.

`PRE`	Precondition (not necessarily checked). `/** PRE: Caller holds synch lock …`
`WHEN`	Guard condition (always checked). `/** WHEN: not empty return oldest …`
`POST`	Postcondition (normally unchecked). `/** POST: Resource r is released…`
`OUT`	Guaranteed message send (for example a callback). `/** OUT: c.process(buff) called after read…`
`RELY`	Required (normally unchecked) property of other objects or methods. `/** RELY: Must be awakened by x.signal()…`
`INV`	An object constraint true at the start and end of every public method.

	`/** INV: x,y are valid screen coordinates…`
`INIT`	An object constraint that must hold upon construction. `/** INIT: bufferCapacity greater than zero…`

Additional, less structured documentation can be used to explain non-obvious constraints, contextual limitations, assumptions, and design decisions that impact use in a concurrent environment. It is impossible to provide a complete listing of constructions requiring this kind of documentation, but typical cases include:

- High-level design information about state and method constraints.
- Known safety limitations due to lack of locking in situations that would require it.
- The fact that a method may indefinitely block waiting for a condition, event, or resource.
- Methods designed to be called only from other methods, perhaps those in other classes.

This book, like most others, cannot serve as an especially good model for such documentation practices since most of these matters are discussed in the text rather than as sample code documentation.

1.3.5 Further Readings

Accounts of high-level object-oriented software analysis and design that cover at least some concurrency issues include:

Atkinson, Colin. *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley, 1991.

Booch, Grady. *Object Oriented Analysis and Design*, Benjamin Cummings, 1994.

Buhr, Ray J. A., and Ronald Casselman. *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996. Buhr and Casselman generalize timethread diagrams similar to those used in this book to Use Case Maps.

Cook, Steve, and John Daniels. *Designing Object Systems: Object-Oriented Modelling With Syntropy*, Prentice Hall, 1994.

de Champeaux, Dennis, Doug Lea, and Penelope Faure. *Object Oriented System Development*, Addison-Wesley, 1993.

D'Souza, Desmond, and Alan Wills. *Objects, Components, and Frameworks with UML*, Addison-Wesley, 1999.

Reenskaug, Trygve. *Working with Objects*, Prentice Hall, 1995.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Accounts of concurrent software specification, analysis, design, and verification include:

Apt, Krzysztof and Ernst-Rudiger Olderog. *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1997.