Object-Oriented Programming, Spring 2009

Sasha Goldshtein, sashag@cs

July 19, 2009

This document contains some brief notes for the OOP exam that I've taken while reading the slides of the lectures and TA classes. It doesn't (by any means) represent everything we've learned, and contains no source code whatsoever (if you are desperate to read Java code, go ahead and re-read your exercise submissions). Therefore, it's entirely *your* responsibility if you use these notes and then the exam surprises you. Nonetheless, I've made every effort for these notes to be accurate, so if you find any mistakes please send me an email or write about it in the course forum.

> All problems in computer science can be solved by another level of indirection... except for the problem of too many layers of indirection.
>
> *David Wheeler, Kevlin Henney*

Contents

1 Pillars of Object-Oriented Programming

The following concepts are primary to the understanding of object-oriented development, architecture and design.

1.1 Encapsulation

Encapsulation means extracting a repeating pattern and placing it in one location in the program. Fundamental to the idea of functions and classes, which contain useful functionality that can be invoked from other parts of the application. In object-oriented programming: Grouping methods and attributes into an object, whose state is then modifiable through an interface. The object maintains its state (retains it) unless it is modified by a method invocation. Variables are private and never directly revealed to the outside world; changing the state of an object is done through methods. Related ideas: information hiding (do not disclose private state variables), implementation hiding (do not disclose how methods are implemented or how the state is being retained). Advantages: The implementation of an object has no effect on the rest of the system (Don't Repeat Yourself); the state is decoupled from its representation (flexibility).

Don't repeat yourself.

1.2 Identity

Objects have unique identities (sometimes called *handles*) which are attached to it when it's created. State changes made to the same object do not change its identity. No two objects can have the same handle. The object's handle can be viewed as its unique address in memory, where no other object can be stored, and which doesn't change across the object's lifetime[1]. It's also possible to create more than one reference (alias) to the same object.

1.3 Messages

Messages are the mechanism that is used to invoke a method on an object. Messages require us to know the identity of an object and the signature of the method, which determines the parameters to pass and the value that is returned. Messages can be divided into several sub-types: Informative messages ("something happened, update yourself", e.g. set a robot's location), interrogative messages ("tell me something about yourself", e.g. get a robot's location) and imperative messages ("do something about yourself", e.g. move a robot forward).

[1] This is actually not entirely accurate, as most JVM implementations feature compacting garbage collection, which causes objects to move in memory. No two objects can share the same memory address, but it's possible for an object's address to "change" during its lifetime.

1.4 Classes

Classes are stencils for new objects, containing methods and attributes (variables). Instances of classes are objects that can be used in a program. Additionally, class methods and variables (static methods and variables) are shared across all instances of a class.

1.5 Inheritance

Inheritance means that the behavior of a class (base class) can be retained by another class (derived class), adding more functionality and possibly overriding some of the base class' behavior. Inheritance should be the model of choice if the "is a" test can be applied. For example, a dog *is an* animal, but it's hard to say that a car is a steering wheel. This makes it easy to make inheritance decisions when related to real-world domain models[2].

Modeling inheritance correctly, i.e. the notion that an object has a certain type S iff its type T is derived from S, is also known as the Liskov Substitution Principle (LSP).

1.6 Polymorphism

Contrast over*riding* vs. over*loading*.

Polymorphism means that a method can be reimplemented (overriden) by derived classes (obviously, only instance methods can be overriden[3]); for this to be useful, it must be possible to invoke the derived class implementation through an instance of the base class. For example, you can feed animals without knowing the specific type of animal involved, but the feeding operation might be implemented differently by different animals. Polymorphism requires *dynamic binding*, wherein a method invocation is resolved only at run-time (because it's impossible to determine at compile-time which method should be invoked). Implementing these concepts often revolves around abstract classes and abstract methods (methods without an implementation), and around interfaces. Note that abstract classes should be used when there's a behavior subset that can be *implemented* for all sub-classes, and interfaces should be used when the behavior can only be *specified*, not implemented, and when there is no "is a" relationship between the parties involved[4].

Try to keep your interfaces immutable.

[2] It's not always so easy to make the call. Run this through your head: cupboard is a furniture, cupboard is a cubic object, cupboard is a brown object... It's easy to get the wrong model by following inheritance blindly.

[3] Some other quirks of Java in this regard: When overriding a method with an exception specification, the overriden method must declare the same exception type or a superclass of it; when overriding a method, it's possible to change its accessibility modifier to allow *more* access (not less).

[4] There's also the technical limitation that Java doesn't allow multiple inheritance, so interfaces are the only choice if you need more than one. Another oddity is that Java interfaces can contain constants (public static final fields).

1.7 Genericity

Genericity means defining a type or a method that depends on (is parameterized by) another type. For example, a collection is often generic in the type of elements that it contains. It hardly makes sense to duplicate a collection implementation for integers, floats, employees and strings separately.

2 Design Patterns

Software design patterns are reusable tricks, design and implementation snippets that recur across different software systems and architectures. Design patterns are all about reusing the architecture and design performed by someone else, although if you apply the same design pattern multiple times, it's unlikely that you end up with the exact same design, not to mention the exact same code[5]. This is not code reuse; it's design reuse.

Design patterns summarize decades of experience of other software developers building software that has the following desirable properties:

- Maintainable
- Modifiable
- Extensible
- Flexible
- Portable
- Reusable

Design patterns are usually presented within a certain problem domain (context), and consist of an abstract solution description and a discussion of its impact, its pros and cons. Patterns can be categorized in different ways, one classic approach is to distinguish between *creational* patterns[6] (that abstract away the object creation process) and *behavioral* patterns (which affect interactions between existing objects).

2.1 Factory Method

Defines an interface for creating an object, leaving the sub-classes to decide which class should be created. Usually, the interface would specify an abstract class (or an interface) to be returned from the factory method, and sub-classes would create a specific derivation of that abstract class.

[5] For more information on design patterns, the best source would probably be the GoF book, not the OOP course slides...

[6] Within creational patterns, we can make a further distinction between *class* creational patterns which use inheritance to decide which object to create, and *object* creational patterns which delegate the instantiation to another object.

2.2 Abstract Factory

Defines an interface for creating groups of related objects. This is an extension of the factory method pattern, only there's need for more than one factory method because more than one "type" of class is being instantiated. The implementation of an abstract factory is essentially a class that deals only with the instantiation of other classes. Often enough, the abstract factory itself uses factory methods of *other* objects to perform the actual instantiation.

Pros — The obvious advantage of factories is *decision localization*: Only the factory implementation knows which objects will be created, allowing greater flexibility and decoupling of the client code. Additionally, the Abstract Factory pattern allows grouping of concrete implementations (e.g. if we need to provide menus, scrollbars and buttons for Windows and Mac OSX, then by using a factory class we can make sure that a Windows button isn't mixed with a Mac scrollbar).

Cons — The disadvantage is the usual disadvantage of programming to an interface: If there's a specific subset of functionality implemented only by *some* concrete classes that is not part of the common interface, the factory client can't use it without resorting to ugly tricks. Additionally, every new interface requires changing the interface of the abstract factory, which is generally an undesirable property.

2.3 Singleton

Ensures that a class has only one instance and provides access to it from any location in the program. There are many possible ways to implement this pattern (in fact, it's a common interview question). To prevent multiple instances from being created, the constructor should obviously be made private. As for accessing the object, one of the better implementations involves lazy instantiation: The object is created only when it's first accessed, through the use of a static method that returns the instance or through the use of static field assignment[7].

2.4 Decorator

Defines an augmentation of an object's behavior without resorting to inheritance, through the use of object composition that can be set and changed at run-time. The idea is that deriving from a class is not a run-time decision, which makes it highly unflexible. For example, if we want to "decorate" a window with a scrollbar, it's undesired to create a special class for "window with scrollbar", as this very quickly leads to class explosion. Composition is the way to go in this case, and because we want to emulate the original object's behavior, it makes sense for the decorating object to implement the same interface as its

Favor composition over inheritance.

[7] There are some threading issues to be careful with here, if multiple threads are accessing the singleton for the first time.

aggregated object, and to delegate messages to its aggregated object (note that decorators are applied to *objects*, not classes!).

Java streams are a classic example of decorators.

Composition is a very powerful technique that can allow us to replace an object's behavior at run-time without breaking encapsulation or relying on the implementation details of the base class. Composition has additional advantages, such as keeping classes small and focused on a single task, minimizing dependencies and forcing decoupling.

Decorators provide both "has a" (composition) and "is a" (inheritance) traits, because we want the decorator to have the same interface as the aggregated object but still delegate messages to an actual object. This also means that it's possible to wrap multiple decorators on top of each other, and as long as they all use well-defined interfaces no one will be any wiser.

Cons — Lots of little objects that are sometimes hard to understand, relationships between objects are hard to grasp. There's also the obvious disadvantage we've seen before with regard to programming to an interface and not the implementation.

2.5 Strategy

Defines an algorithm of choice at run-time. For example, the algorithm that should be used to sum numbers, layout windows or display output is typically run-time pluggable, and should be abstracted "behind" an interface. It's often common to see a *null* strategy, which is useful for testing (stubbing).

Pros — Run-time choice of implementation, decoupling. This pattern is a classic way to get rid of conditionals through the use of polymorphism (although remember that somewhere in your program a conditional statement still determines which strategy variant to use!).

Cons — If there is a whole lot of strategies, there is a whole lot of objects.

2.6 Composite

Defines a recursive relationship in which each component simultaneously implements an interfaces and aggregates a collection of components with the same interfaces. One primary use of this design pattern is in GUI systems. A composite object usually performs some work and then delegates additional work to its aggregated components, recursively until the leaves of the tree are reached. Clients are not required to recognize the difference between a container and a single object, because they behave uniformly, and it's possible for a single object to gain aggregated components at run-time[8].

Pros — It's easy to add new components, and there's no need for the client to be aware of the difference between containers and components.

[8] There's also the question of whether child components should have a reference to their parent (this is not strictly necessary)

Cons — It's difficult to restrict the type of components in a composite, leading to a very general design (for example, if specific operations are required from specific sub-components, it might be difficult to reach them if they are buried deep inside a composition tree).

A very big question is whether the operations for managing the aggregated collection of components in a composite belong to the composite class or to the component class. Putting them in the composite class hinders transparency but advocates safety (for example, it might not make sense to add child components to a certain type of component); putting them in the composite class hinders safety but advocates transparency (composites and components are exactly the same thing). It's also questionable whether the list of aggregated components should be stored in the component or composite class, and it's in fact also questionable whether there's even need for a separate composite class at all.

2.7 Observer

Defines a one-to-many dependency between objects, so that when the state of one object changes, the dependent objects are notified automatically. The Observer patterns decouples the event source (subject) from its dependents (observers). The observable object must have some capability for attaching observers and detaching them where the interest in the event has ended[9].

Pros — Minimal coupling between the subject and its observers, including dynamic modification capability (observers can be added and removed at run-time).

Cons — Observers are unaware of each other, so if there's a two-way interaction between the subject and its observers, it's possible for an infinite cascade of events to occur[10].

It's common for the subject to identify itself as well as the event that occurred, as part of the observer "update" interface.

In Java, the *Observable* base class provides subject functionality, including methods for adding, deleting and notifying observers. The *Observer* interface serves as the base class for all observers that want to register their interest in the subject. The problem with this is that *Observable*, as a base class, limits the ability of the object hierarchy to evolve because classes get only one super-class to sub-class. It's possible to solve this problem using delegation (composition) or rolling a custom version of both sides.

[9] It's important to detach observers if the observer's lifetime is shorter than the subject's lifetime, and if observers are constantly being created. Otherwise, a memory leak might manifest.

[10] E.g., consider a nuclear reactor where a "temperature high" event is published to observers. If one observer chooses to raise one of the reactor rods, triggering a state change in the reactor, and another observer reacts to that state change by lowering the reactor rod, the system might end up in an infinite cascade of events.

2.8 State

Defines a means for an object to alter its behavior when its state changes at run-time. Operations that depend on the state of the object are delegated to an external state interface, which has multiple implementations for the various behavioral states the object can assume. In this fashion, all behavior associated with an object's state (as well as potentially the transitions between states, which also assume a more organized fashion[11]) is delegated to other, smaller objects; this reduces the clutter of conditional statements in the original object's class.

The primary disadvantage of the State design pattern is the need for many small objects, especially as the design evolves and complicated business rules are added to the state machine of the object.

2.9 Additional Design Patterns

Additional design patterns are scattered elsewhere in this document. The MVC design pattern which deals with GUI applications is described in Section 8.4.

3 Exceptions

An exception an *abnormal* event that stops the normal flow of the application's execution. When an exceptional condition occurs, an exception (in Java, like many other languages, this is a special object) is generated and *thrown*. The exception usually contains information about the error that occurred, and it can be *handled* (or *caught*) by methods that are on the same method call stack on which the exception was generated[12]. An exception handler is selected to handle the exception if it specified that it can handle this exception type. If there is no appropriate exception handler that handles the exception, the application is usually terminated[13].

Note the word "abnormal": Exceptions are for exceptional conditions!

The primary advantage of using exceptions is that it makes it harder to ignore exceptional conditions: If an exception isn't caught, the result is fatal for the application. Additionally, handling exceptions in several well-defined locations is cleaner and leads to better separation of responsibilities than mixing error-handling code with happy flow logic.

It is also possible to chain multiple exceptions together, which is especially desirable if there are multiple layers (tiers) of complexity in the application. For example, the data access layer might throw an exception which the business logic layer will wrap in another exception before it reaches the user interface layer.

[11] It's possible to handle state transitions within the states themselves (which encourages coupling between states) or within the context object that uses the states.

[12] This distinction is mainly relevant for multi-threaded applications, in which it's entirely possible for two different exceptions to be handled on two different threads of execution, each of which has its own method call stack.

[13] It is in fact *desirable* for an application to terminate if an exception wasn't handled—it's highly risky to continue execution if there's an unhandled exception in the air!

3.1 Types of Exceptions

In Java, there are three types of exceptions, all derived from the *Throwable* class which provides the stack trace, a message string and other details:

Checked exceptions indicate something recoverable has happened; runtime exceptions indicate there's a programming error; errors indicate there's something *really* wrong.

Errors — Errors are problems inherent to the Java execution environment. They are derived from the *Error* class and arise in situations which the application can't normally recover from (or even anticipate). Examples include out of memory scenarios, VM bugs and other problems.

Runtime — Runtime exceptions are errors inherent to the program, but which are usually not recoverable. They are derived from the *RuntimeException* class. These are software bugs that do not directly depend on user input. Examples include cast exceptions, dereferencing null pointers, passing an invalid argument to a method and other problems.

Checked — Checked exceptions are errors inherent to the program, which could be anticipated in advance and handled at run-time. They are derived from the *Exception* class. These are problems that might have to do with user input and that the application is usually expected to handle. Examples include file not found, end-of-file reached, and other problems.

3.2 Exception Specification

Java is very strict about throwing exceptions. If there's any way that an exception can be thrown inside a method, either of the two conditions must hold:

- The potential exception code must be surrounded by an exception handler[14] (*try...catch* block), or
- The method specifies that it can throw this kind of exception (an exception specification, or *throws* modifier).

While it is desirable for exceptions to be handled if possible, only checked exceptions *must* be handled or specified.

3.3 Cleanup

There are some cases in which there isn't even need for an exception handler, and all that is desired is for some code to be executed in the "cleanup" path, to make sure that there are no lingering resources after the exception is thrown and execution leaves the current method. This can be accomplished using a *finally* block, which is guaranteed to always execute (regardless of whether there was an exception).

[14] When handling exceptions, make sure you handle the most derived exception first.

4 Iterators

Iterators are a generic way to access the contents of a collection (aggregate) which contains multiple objects, usually of the same type. Java iterators (through the *Iterator* interface) provide facilities for obtaining the next element, determining whether there is a next element, and removing the current element from the underlying collection[15].

Because an iterator decouples the client from the underlying collection of elements, it's entirely possible to create virtual iterators which are not based on a collection (for example, an iterator that returns all prime numbers in a certain range). In that sense, iterators need not be finite: It's possible for an iterator to continue reporting the "next" element indefinitely.

Iterators can be roughly categorized into the following groups:

Fail-fast Fail-fast iterators work on live data but become invalid immediately when it is modified (throwing a *ConcurrentModificationException* exception in most cases). Modifying the underlying data through the iterator might or might not be supported.

Weakly-consistent Weakly-consistent iterators work on live data and reflect updates and deletes, and do not reflect inserts.

Snapshot Snapshot iterators work on a snapshot of the data (that is not live) and do not reflect any updates. They might be faster and are definitely safer, but they take longer to create (because the snapshot creation takes time) and possibly do not reflect the real state of your data.

To support the iterator pattern, Java features the *foreach* language keyword which allows sequential access to any collection that implements the *Iterable* interface (including built-in arrays).

5 Streams

Input and output are two very common operations in software, and have a lot in common although there is a multitude of ways to perform each. Streams are an abstraction which hides (encapsulates) the specifics of input and output, and describes only the interface through which data (bits or bytes) gets in and out of our software.

The basic stream classes in Java are *InputStream* and *OutputStream*, which are abstract base classes specifying the fundamentals of input and output operations. Both stream classes denote sequential I/O operations, meaning that the actual input and output are performed in a sequence (of one or more bytes)

[15] Not all iterators support the remove operation: Some will throw an exception when it is called and yet others will do nothing at all.

and there's not necessarily a way to "rewind" the stream to a specific position[16]. This allows for a very rough generalization of all input and output algorithms:

1. Open the stream.
2. While there is more information available, read it from the stream (or write it to the stream).
3. Close the stream.

5.1 Text and Binary

There are two common ways of looking at data:

Text — Textual data consists of characters, which constitute human-readable text. For example, a text file that contains a Shakespeare play can be opened by any standard text viewer such as Windows Notepad. Even though text seems simple, there are actually lots of ways to encode text to byte sequences. Java uses Unicode, which converts every character to a 2-byte representation[17].

Binary — Binary data consists of a sequence of bytes which has no predefined structure. It is up to the application to decide how the data should be interpreted. For example, when storing a sequence of temperature samples, we might want to store 1 byte for each sample, representing a temperature sample in the range $0 \ldots 255$ (this would occupy less space than if we stored the textual representation of these numbers, which might be up to 3 characters, consuming 6 bytes in the Unicode representation). Opening this kind of sequence in a text viewer will not display the numbers in their textual form[18]!

5.2 Java I/O Classes

Java provides two special abstract base classes for dealing with text (character) data: *Reader* and *Writer*. Derived classes take care of the specific type of stream that is being written to or read from: For example, the *StringReader* class deals with reading character data from an existing string instance; the *FileWriter* class deals with writing character data to a file.

Here are some of the common Java I/O classes:

[16] There are specific ways to do this for specific types of input and output, but the stream abstraction does not necessarily entail this "rewind" capability.

[17] A widely accepted alternative is the ANSI encoding, which associates each character with a 1-byte representation. Unfortunately, there are more than $2^8 = 256$ characters in the languages of the world, and therefore Unicode is gaining popularity across a wide spectrum of modern languages, execution platforms and operating systems.

[18] In fact, even text (in sophisticated formats) is often stored as binary data. For example, the PDF format is a binary format—if you open this file in Notepad you'll see a sequence of garbled characters! It's not very surprising considering that PDF files can contain arbitrary content, including math formulae $V = \bigoplus U_i$ and images.

- Streams for reading and writing on memory and strings: *CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringReader, StringWriter*
- Streams for reading and writing files: *FileReader, FileWriter, FileInputStream, FileOutputStream*
- Decorators for buffering reads and writes (reducing the I/O performance cost): *BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream*
- Object serialization[19] streams: *ObjectInputStream, ObjectOutputStream*
- Streams for reading and writing primitive data types in a structured fashion: *DataInputStream, DataOutputStream*
- Decorators for filtering data that is being read or written: *FilterReader, FilterWriter, FilterInputStream, FilterOutputStream*
- Converters (adapters) between byte-oriented and character-oriented streams: *InputStreamReader, OutputStreamWriter*

Always close your streams.

All stream classes have constructors which open the stream when the instance is created. Some of the stream classes are closed automatically (using the finalization mechanism), but it is advised to close the stream immediately after the application is done using it. First of all, this ensures that system resources are released as soon as possible (e.g. there might be a limit on the number of open files in the system). More importantly, it protects against data loss, because some streams (especially buffered streams) might flush the actual data to the file only when they are closed. Therefore, even though the underlying operating system usually closes open files when the application terminates, it's still possible for data to be lost if streams are not explicitly closed.

Exceptions that arise in the input-output process have a common base class, *IOException*.

Other useful classes include:

- The *File* class supports retrieving information about files or directories, such as their last modification time (metadata, not contents).
- The *PrintStream* class supports formatted string output (using the *printf* method which allows formatting specifies such as *%s*, *%f* etc.). The *Formatter* class allows for similar functionality, decoupled from a specific stream type.
- The *Scanner* class supports reading formatted input, and is especially useful for console applications requiring user input.

[19] Serialization is the process of turning an object instance into a sequence of bytes, and reconstructing an object instance from a sequence of bytes.

- The *System.out*, *System.in* and *System.err* fields provide applications with access to the standard output (usually the console), the standard input (usually the keyboard) and the standard error (usually the console) streams. It's also possible to redirect these standard streams to alternative locations.

5.3 Stream Decorators (Filter Streams)

When reading or writing structured data from files, it's desirable to have customizable buffering and data decoding capabilities. In the Java streams library they are provided through the use of *filters* (which implement the Decorator design pattern)—special decorating streams that do not assume the responsibility of performing actual I/O. These filter streams augment existing stream implementations with additional functionality. As with any use of the Decorator design pattern, it's possible to add multiple decorators on top of each other (chaining), leaving them none the wiser.

The taxonomy of input and output streams contains two major types of streams: Source streams, which connect directly to their source of data (such as *FileInputStream*), and filter streams, which wrap other streams and offer decorating functionality (such as *DataOutputStream*).

The *DataInputStream* and *DataOutputStream* filter streams are two very useful examples of filter streams which provide easy access to structured binary data stored in any kind of stream.

5.4 Serialization Streams

Serialization is the process of turning an object instance into a sequence of bytes (a stream); deserialization is the process of turning a sequence of bytes (a stream) back into an object instance. In Java, serialization is accomplished by the use of special serialization streams: *ObjectOutputStream* and *ObjectInputStream*. These two streams allow an object's data (its instance variables) to be stored in any stream and then reconstructed from it[20].

In order for a class to be serializable, it must implement the empty (marker) interface *Serializable*. This indicates to the Java serialization framework that this class allows its objects to be stored as a sequence of bytes. Note that when an object is serialized, there is a recursive requirement that all its fields be serializable as well (primitive types already support this, but you have to take care of your own types). If there's a field that you don't want serialized, you can use the *transient* keyword to indicate this.

When an object is written to a stream, the entire graph of references rooted at this object is serialized with it. Cycles are possible in the graph, so they are

[20] Note that there's no requirement for the deserialization to occur within the same application's lifetime, or even on the same machine. In fact, the two primary uses for serialization are transferring an object's information across processes or even physical machines, and storing an object's information in a file for later use.

serialized as well; when the object is deserialized, it brings back to life the entire graph of references with it.

When an object is deserialized, it's possible that its class (type) can't be found, which will throw an exception. It's also possible that the class was changed in the meantime, which could result in undefined behavior. To make sure that versioning is handled in an organized fashion, you can add a private static variable called *serialVersionUID* and update it whenever you change the source code. This way, deserialization will fail unless the serialized object matches precisely the class that is currently present in the class path.

6 Generics

There are two orthogonal techniques to achieve reuse and extensibility: The first, through vertical abstraction and specialization, i.e. inheritance and polymorphism; the second, through horizontal type parameterization, i.e. genericity. The most typical example is collections—a collection can be modeled as a container of *Object* instances, but then we have to sacrifice static type safety (because it's suddenly possible to add a book to a collection of fruits). Alternatively, a collection can be modeled as a separate independent class for each type of element, but then we have to duplicate a lot of our work, changing only the element type. Type safety is a requirement that can't be easily neglected, and therefore a *generic* but still type-safe solution is sought.

A generic type (or method) accepts additional type parameters (generic parameters). For example, a collection would have a type parameter describing the type of its elements; a comparator interface would have a type parameter describing the type of elements to compare; a sorting method would have a type parameter describing the type of elements to sort; and so on. When using generic types, the client of the code must provide the type argument before instantiating the type. This creates a closed generic type that can be instantiated, and the compiler takes care of type safety so that books aren't added to a collection of fruits.

6.1 Generics in Java

Java generics are different from their implementation in other languages (e.g. C++ templates and .NET generics). In Java, there is only a single compiled version of the generic type, that undergoes a process called *type erasure* such that all type parameters are removed and replaced by *Object*. This implies that it's impossible to provide a primitive type as a generic type parameter (so we can have a *List<Integer>* but not a *List<int>*). Fortunately, the reference wrappers for primitive types (*Integer*, *Double*, *Boolean*) make this easy to reconcile[21].

Note that because type parameterization is orthogonal to sub-classing, there is no inheritance relationship between different instantiations of the same generic

[21] ...although incurring a run-time performance cost for boxing and unboxing, and occupying significantly more memory than their primitive counterparts.

type. For example, it's impossible to assign a *List<String>* to a *List<Object>* variable, for type-safety reasons. If it were possible, we could insert books into a collection of fruits, violating the very fundamental principle that we wanted to implement!

Generic types are not related by inheritance.

Because of the type erasure process, within the implementation of a generic type there are only very few operations that can be performed on a naked type parameter. For a type parameter *E*, these include:

- Assignment to another instance of *E* (or from another instance of *E*)
- Equality comparison with another instance of *E* using == and !=
- Parameter passing to a method that accepts *E* or *Object*
- Method call of any *Object* method (e.g. *toString*)

While naked (unbounded) generic parameters are nice, they obviously aren't enough[22].

6.2 Wildcards

Because different instantiations of generic types are incompatible with regard to assignment or parameter passing, there must be a way to declare that a method accepts multiple versions of the same generic type (before generics, this problem did not exist because there was no such thing as multiple versions of the same type that are not related by inheritance). For a generic type *MyType<E>* we declare another open type that can be assigned any specific instantiation such as *MyType<String>* and *MyType<Integer>*, and that type is called *MyType<?>*. This is called a wildcard.

Regardless of the actual generic parameter that is used, at run-time we can refer to the wildcard as *Object*. For example, if we have a wildcard generic collection parameter, we can extract elements from the collection and call the *toString* method which is common to all objects. However, in the same example, we can't add arbitrary instances to the collection (not even *Object* instances!) because we don't know the element type of the collection.

Bounded wildcards extend this idea even further by specifying that although the generic parameter isn't completely known, it is bounded from above in the inheritance hierarchy. For example, we can specify a wildcard that would match any type derived from *Shape*. This kind of wildcard is written *MyType<? extends Shape>*[23].

[22] There are some additional limitations imposed by the environment. For example, it's impossible to instantiate a generic array (i.e. *E[]*) in a type-safe fashion (only using reflection), because after type erasure instantiation would be ambiguous at runtime. Another limitation is that an inner class that wants to use the generic parameters of its enclosing class must be non-static, or redeclare the parameters.

[23] Note that this doesn't help with the collection scenario. If we have a collection of *? extends Shape* we still can't add anything to the collection because we don't know the element type of the collection. However, we can extract *Shape* objects from the collection and invoke *Shape* instance methods on them.

The symmetric concept is the idea of a wildcard that is bounded from below in the inheritance hierarchy. For example, we can specify a wildcard that would match any type that is a super-class of *Shape*. This is useful in various scenarios, and most commonly encountered with comparators: If we have a sorted collection of *E*, then we need a *Comparator<E>* to sort them. However, this is somewhat limiting: We should rather use a *Comparator<? super E>* to sort our elements (if it can sort shapes, it can certainly sort squares, but of course not vice versa). Note that when using lower bounded wildcards the collection scenario we've seen before is reversed: If we have a collection of *? super Shape* then we can add shapes, squares and triangles to it; however, when extracting elements from the collection we can assign them only to *Object* because there is no bound on their type[24].

Type bounds can be recursive, which is slightly misleading but extremely useful. For example, if we want a method to be used only with types that can be compared to themselves, we can specify *T extends Comparable<T>* as a bound. The more contrived *T extends Comparable<? super T>* is even more flexible.

6.3 Runtime Details

The type erasure process implies that there's only one *Class* object for all generic type instantiations, so that calling *getClass* on a list of strings or a list of integers would produce the same result. It also means that static variables and methods are shared across all generic type instantiations, so you can't refer to type parameters in a static method. Finally, the most subtle point is that casting to generic types is meaningless because there is only one runtime type for all generic type instantiations.

It is possible to refer to the *raw* generic type, which doesn't include the generic parameters (for example, if we have a *List<E>* we can refer to *List* as the raw generic type). Referring to the raw type is unsafe and generates a compiler warning; usually, it's better to refer to the type parameterized with *Object* instead (e.g. *List<Object>*).

6.4 Generic Methods

Generic methods are an interesting alternative to generic types when there's no need to declare a full type for a certain operation. It's common for generic methods to use wildcards, but these can usually be replaced with bounded generic parameters. For example, we can define a method that copies a *Collection<T>*

[24] The discussion in this section is also known as generic covariance and contravariance. For two types *D extends B* we say that *C<X>* is covariant if *C<B>* is assignable from *C<D>*, we say it's contravariant if *C<D>* is assignable from *C<B>*, and we say it's invariant if neither condition holds. (We can also say it's bivariant if both conditions hold.) Oddly enough, Java arrays are in fact covariant, leading to contrived run-time errors, e.g. if you assign a string array to an object array variable and then attempt to store an integer in the array using the standard array access operator.

to a *Collection<? super T>*, but we can also define a method that copies a *Collection<T>* to a *Collection<S>*, where the constraint *<T,S super T>* appears in the method declaration.

The compiler infers type parameters to generic methods, so there's usually no need to specify it (even if the generic method is parameterized only in its return value—in that case, the left-hand-side of the assignment will allow the compiler to infer the method parameter). Only when there's an ambiguity the compiler will signal an error and require the type parameters to be explicitly indicated when calling the method[25].

7 Inner Classes

In Java, it's possible to declare multiple kinds of inner classes. Classes can be nested within a class, within a method or even specified without a name (anonymous classes). Even interfaces can contain nested classes. Nested classes can also be static or non-static. In this section we will briefly examine the differences and use cases for these language features.

7.1 Nested Classes

A class can be declared inside another class, and have an accessibility modifier declared like any other class member (unlike top-level classes, which have only two applicable accessibility modifiers). Nested classes gain automatic access to the private members of their enclosing class, and the enclosing class gains automatic access to the private members of its nested classes.

Nested classes can be static, meaning that they don't have access to the instance of their enclosing class and are created independently of it; or they can be non-static (sometimes called *inner classes*), meaning that they have access to a specific instance of their enclosing class and are created only through that instance that binds them together. Inner classes can't declare any static members.

7.2 Local Classes

Local classes are declared within a method (or a variable initialization block inside a method). They are addressable by name only within the scope of the method in which they are declared, and therefore they can't have any accessibility modifiers or declare static members. It's also impossible to declare local interfaces—only classes can be defined locally.

Like nested classes, local classes gain automatic access to the members of their enclosing class, but they also gain access to the local variables of the method in which they are declared. The only limitation is that these variables

Local classes have access to final local method variables.

[25] The syntax for doing this is a little bizarre: *Collections.<Object>singleton(10)* will call the *singleton* static generic method of the *Collections* class and specify that the generic type parameter is *Object* (in this particular case, this means that the return value will be

must be declared *final* for the local class to access them. This is an important limitation that also applies to anonymous classes, and the reason for it is that the lifetime of the local class instance might be longer than the lifetime of method local variables. Because of that, local variables from the method are cached within the local class generated by the compiler, and to ensure that they remain consistent with their values in the method body, they are required to be final.

7.3 Anonymous Classes

Anonymous classes don't have a name at all, and are defined within a single expression using the *new* operator. Anonymous classes can't explicitly implement interfaces or extend a super-class; the type name specified in the invocation of operator *new* determines whether the class will derive from the specified super-class or will implement the specified (single) interface.

Having no name, anonymous classes can't define a constructor[26]. Similarly to local classes, they don't have any accessibility modifiers and can't declare static methods.

8 Graphical User Interface (GUI) Programming

An application is said to be *event-driven* if it enters idle states and reacts to a set of predefined events. Such applications are also called *reactive*. Classic examples of reactive applications are the operating system (which usually sits idle and waits for external interrupts) and graphical user interface (GUI) applications (which usually sit idle and wait for user input). Event-driven applications usually don't have a well-defined termination time: They execute indefinitely, and their flow of execution is determined by external factors and not by the program itself.

Other desirable properties include concurrency and non-blocking event handling.

Simulations are useful and even critical in certain areas of computer science research, and may be equally useful during development. Simulations consist of the following components:

State — State variables determine the current state of the simulation, and allow it to be persisted, stopped and then restarted. State variables can be discrete or continuous.

Event — Events drive the system from one state to another.

Time — The time model (continuous or discrete) determines whether the system is always well-defined, or whether there are leaps between discrete points in time.

Set<Object>).

[26] Nonetheless, it is possible to pass parameters to the super-class constructor, using the *new* operator invocation. For example, *new Shape(5) { ... }* specifies a new anonymous class derived from *Shape* that passes the number 5 to the super-class constructor.

Deterministic Simulations can be deterministic (always producing the same result given the same state) or probabilistic (producing a possibly different result given the same state).

Trigger Changes in the simulation can be triggered by a time-line which updates the system state, or triggered by external events.

8.1 Event-Driven Design

Designing an event-based system usually begins with defining the *events* that can occur in it, and which drive the system from one state to another. Events are usually enqueued into a priority queue, and dequeued from the queue by the application that handles them[27]. Often enough, there is a specialized event dispatcher that dequeues events from the queue (periodically or by a trigger) and dispatches them to the appropriate components in the application.

8.2 GUI Systems

A graphical user interface (GUI) application consists of components on the screen (sometimes called *widgets* and *controls*) which are organized hierarchically and accessed using the keyboard and mouse. Some examples of controls: Button, menu, status bar, combo box.

There is a natural containment relationship between GUI components. For example, an Internet Explorer window has a border, a title bar (caption) with minimize, maximize and close buttons, and an inner frame. That inner frame contains multiple tab controls, and each tab control contains a variety of other controls as you visit various web pages. Because of the containment hierarchy, it's possible to easily move an entire subset of controls from one location in the GUI to another.

It's usually the responsibility of the windowing environment and/or the underlying operating system to move windows around, resize them, position them and control keyboard and mouse input focus. It's usually the responsibility of the application to react to other user interface operations, such as button clicks, list box selections etc. The operating system and the windowing environment deliver events (such as mouse clicks and keyboard input) to the application. The windowing environment gets a first look at the event (for example, it could be a close operation that can't be intercepted by the application) and if necessary, dispatches it to the appropriate application for processing. The application sits idly in a loop waiting for events to be delivered[28]. These events are usually handled by the specific widgets which ought to receive them (for example, if the user clicks a button then it's reasonable for that specific button to handle the event, potentially escalating it to other registered observers within the application).

GUI applications are reactive.

[27] For example, the Windows GUI paradigm is all about the operating system (and other applications) enqueueing *window messages* into the queue of another application that dequeues and handles them.

[28] It's also possible to interleave this with background processing by using multiple threads of execution.

8.3 GUI in Java

Most of the GUI system is already implemented in the AWT, JFC, Swing and other frameworks. They define an architecture, rules of conduct and a great variety of useful widgets for GUI applications to use. GUI programming usually involves two activities:

Specializing Specializing widgets for the specific task at hand by sub-classing existing widgets or by using composition (aggregation).

Extending Extending widgets by writing event handlers that register for notifications when a specific condition (such as a mouse click) occurs within a specific widget.

The GUI widget framework uses the Composite design pattern extensibly. Almost all widgets support composition and implement common interfaces for common operations such as drawing, refreshing, updating and other UI-related work. This ensures that components need not be aware of the specific aggregates they are using, and maintains the Open-Closed Principle (OCP).

8.4 Model-View-Controller (MVC)

Model-View-Controller is a design pattern or architectural pattern specific for GUI applications. The various parts of a GUI applications can usually be categorized as follows:

Model The data that is being manipulated.

View The graphical interface that displays the data and provides manipulation capabilities through user input. When the model changes, the view should usually be updated.

Controller The logical entity that coordinates the interaction between the model and the view. Specifically, the Controller handles user actions and translates them to interactions with the Model state.

The model and the view are obvious and present in almost any GUI application; the Controller is slightly less intuitive. The primary reason for the controller to exist is to decouple the relationship between a specific model and a specific view. This allows the same model to be displayed using multiple views.

Controllers decouple Views from Models.

In Java's Swing, the view is a Swing widget (such as a *JButton*), the controller is an *ActionListener* (or a class that is registered through a listener[29]) and the model is an ordinary Java class, collection or database[30]. In this architecture, the view has a reference to the controller, and the controller has a reference to both the model and the view. The view notifies the controller of

[29] The most common way to register for notifications is by implementing the *ActionListener* interface, which contains a single method called *actionPerformed*.

[30] There are also additional listener interfaces, such as *MouseListener*, *MouseAdapter* and others.

events that occur in the system, and the controller interacts with the model and updates the view accordingly. Sometimes, the model has a reference to the controller as well, to effect changes in the view when the model is changed.

An extended version of the MVC pattern includes the Observer design pattern. The view can be an observer of the model, so that if the model changes independently of the user's actions, the view can still be updated. Because a model might have multiple views, the Observer pattern handles the problem of notifying all registered observers (views) when the subject (model) changes. The notifications usually do not carry the entire list of updated information, because it's easy enough for the views to query the model for the changes relevant to them[31].

When this extended version of MVC is implemented, the controller becomes responsible for coordination work only (which works well with the Single Responsibility Principle—a class should do one thing and do it well). If an action that originated from the view has caused the controller to change the state of the model, it is expected from the model to update the view—it's no longer the controller's responsibility.

8.5 Practical Swing

Swing components include *windows*, *panels* and other stand-alone components. Windows contains frames; frames often contain panels; panels contain components. Components might contain additional components—the nesting is unlimited thanks to the Composite design pattern. Layout managers dictate how components are visually laid out on the screen.

The most common sequence of operations is to create a component, initialize it by calling setter methods, add it to a panel or frame and (optionally) add listeners to receive notifications when user actions occur.

The following is a list of some of the Swing classes:

Controls	*JCheckBox, JComboBox, JList, JTextField, JToolTip, JRadioButton, JPasswordField, JSlider, JTree, JTextArea, JLabel, JButton, JProgressBar, JSeparator, JSpinner, JTable, JMenu*[32]
Containers	*JPanel, JSplitPane, JScrollPane, JLayeredPane, JTabbedPane, JToolbar, JInternalPane*
Layout	*FlowLayout, GridLayout, BorderLayout, CardLayout, GridBagLayout*

Most Swing GUI elements can be used without an explicit model and an explicit controller. However, even the simplest GUI applications often approach very quickly a level of complexity that requires a more thoughtful separation

[31] For example, it might be the case that views are interested in different subsets of the updated information. A graph view might not care about the level of detail that a spreadsheet view requires.

[32] *JMenu* is actually positioned outside the content pane, immediately beneath a top-level container.

of concerns. For example, the Swing *JList* class presents a list box which can rely on a static model (e.g. an array), a *DefaultListModel* implementation, or a custom *ListModel* implementation which provides the items on demand and notifies the view when the model changes.

Swing dialog windows can be presented in a modal or modeless fashion. Visible modal dialogs block input to the rest of the application and require immediate user input before proceeding (for example, it might make sense to make a File Save dialog modal). Modeless dialogs do not interfere with the rest of the application. Displaying dialogs is done using the *JOptionPane* class or by sub-classing the *JDialog* widget[33]. To show a dialog, use *setVisible(true)* and to close it use *setVisible(false)*. Additional special dialogs are available, such as *JFileChooser*.

8.6 Data Binding

Data binding is a special case of the Observer design pattern, where some data (in the model) should be dynamically bound to the view. Data binding means that when a property of the data changes, the view is updated accordingly. Because the most common scenario is a change of a property, special types including *PropertyChangeSupport* and *SwingPropertyChangeSupport* provide the necessary infrastructure. Through *PropertyChangeSupport* objects, observers can register their interest in property changes and subjects can publish property change notifications including the name of the property that has changed.

Data binding and concurrency were not covered in depth.

8.7 Concurrency

A well-written GUI application doesn't freeze. Freezing renders the application incapable of handling user input, and is considered unacceptable. This means that background tasks (any operation that does not complete immediately) must be scheduled in the background without affecting the event-handling loop.

Java's Swing uses an Event Dispatcher Thread (EDT) to perform the updates of on-screen elements. It's the only thread from which painting and UI updates can legally occur. To schedule asynchronous background work in Swing, use the *SwingUtilities.invokeLater* and *invokeAndWait* methods.

9 Collections

Collections (or containers) are objects which aggregate multiple elements and provide some of the following operations:

[33] When sub-classing *JDialog*, it's common to intercept dialog closing by using the dialog's *addWindowListener* method, providing a *WindowAdapter.windowClosing* implementation. This allows dialogs to call back to the component that created them. An alternative is to store the result information in a field that can be accessed by the dialog's creator (this is especially easy if the dialog is modal).

- Storing elements in the collection (including dynamically adding new elements);
- Retrieving elements from the collection (by key, by value or by index);
- Manipulating collection elements (such as sorting or aggregating).

Not all collections support all of the above operations, and some collections provide special semantics that others don't. The Java Collections framework contains interfaces that represent collections independently of their implementations, classes that implement collections (reusable data structures) and algorithms that perform useful polymorphic operations (e.g. sorting) on different implementations of collection interfaces.

Using an existing framework is almost always a good idea, even more so in the case of the Java Collections framework. Reusing existing collections reduces programming effort, and ensures that you're using a high-performance, reliable and interchangeable collections implementation.

9.1 Collections Interfaces

The core collections interfaces are generic, and there is an inheritance hierarchy between the various interfaces. Some of the methods dictated by the interfaces are designated *optional*, meaning that a particular implementation might choose to not support them (document this fact and throw an *UnsupportedOperationException* exception in the implementation).

Collection<E> The root of the collections hierarchy, representing a group of elements. There are almost no requirements of a direct implementation of this interface. A collection can be iterated (extends *Iterable<E>*[34]) and can report its size, search for an element, convert the collection to an array[35], and add and remove elements (optional). Additional bulk operations are specified, including *containsAll*, *addAll*, *removeAll*, *retainAll*, and *clear* (all optional except *containsAll*). Finally, collections are also expected to provide at least two constructors: An parameterless constructor for initializing an empty collection, and another constructor with a single argument of type *Collection<E>* which creates a new collection with the same elements, sometimes called a conversion constructor. (Elements are usually shallow-copied.)

Bulk operations are more convenient and usually more *efficient*.

[34] Iterators support the *hasNext*, *next*, and *remove* (optional) operations. The *remove* method is the only safe way to modify a collection while iterating it. Other means of modification might cause the iterator to throw a *ConcurrentModificationException* exception when it's next accessed.

[35] Two conversion methods are available: The first overload returns an array of objects, and the second overload returns an array of a specified generic type. Using it is slightly counter-intuitive: If you want back an array of strings, use: *coll.toArray(new String[0])* and you'll get back a newly allocated *String[]*.

Set<E> Derived from *Collection<E>*. A collection that cannot contain duplicate elements (after the mathematical abstraction). An implementation of this interface is not necessarily ordered in any way. Additionally, implementation of this interface are required to override *equals* and *hashCode* so that two set instances can be compared even if they are not implemented by the same collection class (specifically, two set instances are equal iff they contain the same elements).

SortedSet<E> Derived from *Set<E>*. A set that maintains a sorted (in ascending order) collection of elements. Additional operations for ordered data are present in this interface. Implementations require a means of comparing elements (so that they can be sorted). This interface provides range view access to elements (*subSet*, *headSet*, *tailSet*) and endpoint access (*first*, *last*). Additionally, the iterator is guaranteed to return elements in sorted order and the array conversion operator is guaranteed to return an array in sorted order. While range views on lists do not work so well for a long time because they break if the list is modified, range views on sets write back changes to the underlying set and vice versa.

List<E> Derived from *Collection<E>*. An ordered sequence of elements, providing random-access by index (*get*, *set*), control over where elements are inserted, range-view iteration capabilities (*subList*), as well as search which returns the index of the searched element (*indexOf*, *lastIndexOf*). Lists can contain duplicate elements. Modifying the list through the range view is supported but the range view will break if the underlying list is modified independently of the range view.

Queue<E> Derived from *Collection<E>*. A collection used to hold elements waiting for processing. Queues are usually FIFO, but there are exceptions (notably, priority queues). A queue has a head, and FIFO queues perform insertions at the queue tail. Queues offer two flavors of each fundamental operation: One that throws an exception if it fails, and another that doesn't. The three pairs of operations are the following (the first in a pair throws, the second returns a meaningful error value): Insertion is performed using *add* or *offer*, removal is performed using *remove* or *poll*, examining the top element is performed using *element* or *peek*.

Map<K,V> An association of keys to values (key-value pairs) that cannot contain duplicate elements, meaning that each key can map to at most one value. Implementations do not require any sorting order on the elements. Hash tables are a typical map implementation. Note that this interface is *not* derived from *Collection<E>* and allows searching for elements based on key (*containsKey*) and value (*containsValue*), although the former is usually much faster. It's also

possible to obtain a collection view of the keys (*keySet*), the values (*values*) and the pairs of entries (*entrySet*).

SortedMap<K,V> Derived from *Map<K,V>*. A map that maintains its key-value pairs in ascending sorted order of the keys (values, on the other hand, are not sorted). Implementations require a means of comparing keys (so that they can be sorted).

The Java collections framework contains several skeletal (abstract) implementations of the above interfaces, which are often incomplete and inadequate but serve as a basis for collection development. Among them you'll find *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractQueue* and *AbstractMap*. If you're developing your own collections, you might find them to be of value.

9.2 Collection Implementations

The following is a list of implementations of the above collections interfaces:

Set — Set implementations include *HashSet<E>* which uses a hash-table and makes no iteration order guarantees, *LinkedHashSet<E>* which uses a threaded hash table and guarantees iteration in order of insertion, and *TreeSet<E>* which uses a red-black tree and requires ordering on the elements. One of the most useful operations with a set is duplicate elimination.

List — List implementations include *ArrayList<E>* which uses a dynamically expanding array, *LinkedList<E>* which uses a linked list and *Vector<E>*. Because lists are ordered, useful operations include the ability to concatenate lists without losing the relative order of elements within each list. Another useful operation is *Arrays.asList* which allows an array instance to be viewed as a list without copying it. Obviously, this method doesn't allow modifying the size of the list (because arrays are not resizable). List iteration can be done using the standard *Iterator<E>* interface, but the *ListIterator<E>* interface is more powerful and allows traversing the list backwards (using the *hasPrevious* and *previous* methods). Some useful algorithms applied to lists include *Collections.sort*, *shuffle*, *reverse*, *rotate*, *swap*, *replaceAll*, *fill*, *copy*, *binarySearch* and others.

Queue — Queue implementations include *PriorityQueue<E>* which sorts elements according to their natural order (and doesn't use FIFO order) and *LinkedList<E>* which maintains FIFO semantics.

Map — Map implementations include *HashMap<K,V>*, *LinkedHashMap<K,V>* and *TreeMap<K,V>* which mirror similarly named implementations of the *Set<E>* interface.

9.3 Advanced Usage

The Java Collections framework does not include an implementation of a multi-map, in which each key can be associated with multiple values. Fortunately, it's easy enough to mimic this behavior using a standard map whose values are list instances.

To nicely fit in the view of the world imposed by ordered collections, types should implement the *Comparable<E>* interface or provide an instance of the *Comparator<E>* interface to collection constructors. However, even the unordered collections will benefit from a proper implementation of *equals* and *hashCode*, two methods defined in *Object* which are relevant when objects are compared for equality and it's possible for two distinct object instances (having different identities) to be equal based on their value. Here are some guidelines for implementing a proper type that is embedded in a collection:

Equality, hash code and comparison must be consistent with each other.

- Make the type immutable if possible (makes life easier if the object is used as a key in a set or a map);
- Check arguments in the constructor to ensure the object is valid to begin with;
- Override the *hashCode* method so that it's consistent with the *equals* method (two equal objects must have equal hash codes);
- Override the *equals* method and make sure it checks if the parameter is null or is of a different type, in which cases a runtime exception is thrown;
- Override the *toString* method to facilitate diagnostic output and debugging in an IDE;
- Implement *compareTo* and ensure that your implementation maintains anti-symmetry, transitivity, totality and consistency with the *equals* method, which are the requirements from a total ordering of the elements.

Additionally, if you implement a comparator, it's easy enough to use for sorting of lists and similar unordered collections, but it might not work properly if you're using it as a comparator for a total-ordering collection such as a set. The reason is that your custom comparator must still implement a total ordering, so for example if you want to sort employees by their salary you must still ensure that your comparator does not return 0 for two different employees which happen to have the same salary.

There are also some useful collection wrappers that will augment an existing collection instance with additional semantics, such as multi-threaded synchronization of access, locking a collection for modification (read-only) and similar operations. These are static methods of the *Collections* class, such as *Collections.synchronizedSet* or *Collections.unmodifiableList*.

The static *Collection* class also contains some factory methods for creating typical collections, e.g. *Collections.emptySet*, *singletonList*, *nCopies* and others.

9.4 Choosing a Collection

The criteria for choosing a collection to use are usually the following:

Duplicates If you require a collection that supports duplicates, you can't use a set or a map.

Ordering If you require a collection that is always sorted, use a sorted set or a sorted map.

Complexity Consider the complexity of individual operations to see which one you need most often (e.g. for quick searching use a hash map, for quick inserts use a list etc.).

There are also other, secondary criteria, such as support for null values, support for synchronization (multi-threaded access) and other considerations.

10 Object-Oriented Design Principles

In this section we will look into some principles of object-oriented design.

10.1 Modularity

A modular design should satisfy the following requirements:

Decomposability The design should decompose a problem into a small number of simpler sub-problems, on which work can proceed independently. Once this has been done, it should be possible to distribute works across these new modules to different groups of programmers. (This is the typical purpose of top-down design.)

Composability The design should produce software elements which can be combined to form new systems, possibly different from the ones they were intended for in the first place. The key to this is autonomy of individual modules, so that they can be usable in different contexts independently of other modules in the same system.

Understandability The design should allow a reader to examine a module without having to learn about all other modules in the system. This supports maintanability.

Continuity The design should be stable enough so that a small change of specification will result in a design or implementation change of only a small number of modules (preferably a single module). This is also known as Don't Repeat Yourself: If you make decisions in only one place, it's unlikely for any one specification change to affect multiple independent modules in the system.

Protection The design should accommodate for exceptional conditions so that they remain confined to the same module, or leak to at most a small number of neighboring modules.

Modularity is all about two important design principles which are fairly easy to grasp:

OCP The Open-Closed Principle dictates that software entities should be open for extension but closed for modification. In other words, design modules that never change! When requirements change, it should be possible to extend behavior by adding new code (extension) instead of modifying existing, working code (modification). The usual means for accomplishing OCP is through the use of abstraction and programming to an interface: When programming to an interface, changes of the underlying implementation of the "other side" do not effect any change of the existing, working code. Inheritance, polymorphism, and generic types are all means to the end of OCP[36]. Related principles are encapsulation of member variables, dismay at the sight of global variables, and use of polymorphism instead of run-time type information (*instanceof* and its ilk).

SCP The Single-Choice Principle dictates that only one module in the system must know about each individual choice. In other words, for each decision there should be a *switch* or *if* statement in a single place in the program. This allows for great flexibility and extensibility, because adding additional options to the choice means modifying only a single location. SCP is all about limiting the amount of information that is available to modules in the system, which can be viewed as a form of inter-module information hiding.

10.2 Program to an Interface and not an Implementation

Class inheritance is a means of accomplishing reuse, but also a means of adhering to an interface. As long as client code is written against the interface and not a specific implementation (e.g. *Drawable* instead of *YellowCircle*), clients remain blissfully unaware of specific types of objects and are shielded from future changes in the underlying implementation. This produces a less brittle design because implementations can be swapped without affecting the client, reducing inter-module dependencies.

Creational design patterns are usually the way to go for opting into a concrete implementation of the interface the client is working with. Abstract Factory, Singleton and other patterns are all about making the decision of what concrete class to return to the client (SRP).

Abstract classes and interfaces are both means of programming to an interface instead of an implementation. Abstract classes might provide greater

[36] Obviously, it's impossible to close a program against all possible changes, so this should be a design decision.

flexibility (because you can modify an abstract base class implementation without breaking existing sub-classes), but interfaces provide freedom with regard to the base class (because Java doesn't support multiple inheritance).

10.3 Inheritance vs. Composition

The two most common techniques for reusing functionality are inheritance and composition. Inheritance (also called white-box reuse) is a class reuse mechanism, while composition (also called black-box reuse) is an object reuse mechanism.

Inheritance is fairly easy to use, and is supported by the programming language. Overriding existing behavior is also easy through the language constructs. Unfortunately, the static nature of inheritance (the fact that its behavior can't be changed at run-time) is a major disadvantage. Additionally, inheritance encourages tight coupling between super-classes and sub-classes, forcing each one to cringe under the other's change.

Favor composition over inheritance.

Composition is somewhat harder to use, and usually tends to create scenarios with more objects and more complicated object relationships (bits of behavior scattered across the system). However, composition has the advantage of being fully dynamic and customizable at run-time, and reduces implementation dependencies (coupling) between objects. This helps classes remain focus on a single task.

It's usually desirable to choose composition over inheritance[37], whenever possible. It shouldn't be required to create a new class just to achieve reuse—composing the functionality of existing components should often be sufficient. A very strong means of combining the power of inheritance with the flexibility of composition is the use of *delegation*, where an object delegates requests to another object it aggregates (this is the idea behind the Strategy design pattern). This makes it easy to compose behaviors at runtime, at the cost of increased complexity.

11 Multi-Threading

Multi-threaded (or concurrent) execution is about doing multiple things at once within a single application. Whether "at once" is an illusion or not, the need to multiplex several paths of execution at once stems from many reasons.

Among them is *parallelism*, the ability to take advantage of multiple CPUs and multiple processing cores on each CPU, and *responsiveness* (asynchronous processing), the ability of an application to maintain the appearance of being responsive to the user's commands while at the same time performing background processing[38]. Due to the fact that much of the work performed by modern ap-

[37] Composition is also often more easily testable (through dependency injection, inversion of control and other techniques) than inheritance. However, we didn't address testability problems in class or in these notes.

[38] Some would also say that threads lead to a more modular design, or that they provide a better model of the problem domain. This is often not true simply because threads are so confusing to program.

plications is input and output, which does not require the CPU to be involved, it's also possible to multiplex CPU activity with I/O activity using concurrent execution.

Instead of *faster* processors, future computers will have *more* processors.

Concurrent execution is significantly harder than serial execution. It's harder to develop, understand, debug and maintain. Nonetheless, nowadays it's nearly impossible to conceive an application that does not require multi-threading at least to some extent.

11.1 Processes and Threads

A process is an instance of an executable program that is currently executing in the computer's memory. A thread of execution within a process is a flow of control. Multiple threads within the same process share the address space of that process, and have the ability of running simultaneously if there are multiple processors, or virtually simultaneously if there's only one processor. Processes are isolated from each other; threads are not. Threads have a private stack (implying that they have private local variables), and this is what allows them to be used for concurrent execution.

The operating system is responsible for scheduling threads for execution. Only one thing can be executing on a single processor at any given instant of time; therefore, the operating system must preempt executing threads and schedule other threads to replace them on the processor. If the operating system is running on a machine with multiple processors, this scheduling process repeats itself for each physical processor present on the machine. Although the operating system scheduler is the ultimate decision maker with regard to scheduling, we can use threading APIs to affect the way scheduling is performed.

Multiple threads of execution within a process are not a panacea: They have their own problems. In the subsequent sections we will examine the advantages and disadvantages of using threads.

11.2 Threads in Java

Java threads are represented by the *Thread* class, which provides the facilities for creating threads and controlling their execution. Every Java program is started with a main thread, and there are additional threads created by the JVM for its own purposes, including garbage collection and finalization. (There are also alternative approaches to thread creation in Java, including using an execution service—a thread pool—for scheduling threads.)

Threads have their own call stack and local variables. Exceptions propagate through the call stack of the thread on which they were thrown; uncaught exceptions terminate that thread of execution only. Multiple threads can share access to data (allocated from the heap) and code.

To create a new thread, write a class that derives from *Thread* or implements the *Runnable* interface. In both cases, what you really need to do is to override the *run* method and specify what code should be executed within the thread. That method serves as the entry point (the "main") for that thread, and when it

returns—the thread terminates. To start the new thread, use the *start* method (do not call *run* directly, as it will execute the code of the thread within the current thread).

The Java application terminates when all the threads within it have terminated. The main thread might have already ended, but other background threads might keep the application from shutting down.

A thread might choose to yield its execution time to another thread using the static *Thread.yield* method. This does not guarantee that any specific other thread will be scheduled. A thread may also yield its execution time for the specified amount of time using the static *Thread.sleep* method[39].

Sleeping is one way a thread can enter the *blocked* state. When a thread is blocked, it is not contending for execution—in fact, it is not ready to execute. A thread that is in the blocked state can be interrupted using the *Thread.interrupt* method. This sets the interrupted status flag and causes an *InterruptedException* exception to be thrown within the context of the interrupted thread, and wakes it up from its blocked state immediately. Even if the thread is not blocked, it can poll the *Thread.interrupted* method to see if it was interrupted from the outside.

Do not brutally abort threads.

Coordination of work between threads is a matter of great delicacy. It might appear that brutally terminating a thread whenever you're done with it is the way to go. However, it is extremely dangerous to terminate a thread from the outside, especially if you are not sure what work the thread has already completed and what the precise state of the program is at the time of the abort. Interrupting a thread is not a much better alternative; it's best to use a synchronization mechanism (discussed later) or even a global variable that the thread's code can periodically poll to determine whether it should abort its execution.

The *Thread.join* method can be used to wait for another thread to complete its execution. This allows for the creation of multiple threads that run off to do some work, and then meet again at a barrier when they are all done. Threads can be grouped into groups (specified at creation time), which allow for coarse control of an entire execution tree.

Finally, the *Thread.setPriority* method can be used to recommend a priority for the thread to the operating system scheduler. Depending on the JVM and operating system, this might end up as an ignored hint or a binding command.

11.3 Data Synchronization

Threads share the same memory of the process they are executing in. This is convenient, but also highly dangerous as threads might corrupt the data that other threads are using. A thread accessing a global piece of data must ensure that it is doing so in a synchronized manner.

The best approach to take is to avoid the need for synchronization in its entirety. For example, if your data is immutable (read-only or final) then there

[39] Don't use *Thread.sleep* for accurate timing of anything. Among other things, its resolution depends on the JVM, the operating system and the hardware, and can hardly be relied upon.

is no risk of it becoming corrupted. Alternatively, if all your data is private to one (and only one) thread[40], then again there is no problem.

There are several levels of thread synchronization. The most primitive of all is the *volatile* keyword, which can be applied to primitive types and object references. This keyword means that whenever a variable is accessed, it is accessed directly in main memory, bypassing processor caches. This guarantees that all threads (even if running on different processors) see a consistent view of the volatile variable[41].

A higher level view of synchronization is provided by locks. Every Java object (except for primitive variables) is associated with a lock. Each such lock can be held by no more than one thread at a time, and a thread that attempts to acquire a lock that is already taken will transition to the blocked state until the lock is released. When the thread finishes using the lock, it releases it for its peers to use (which wakes up any threads that were waiting to acquire the lock). In other words, only one thread can execute code protected by a lock at the same time; this means that only one thread can access the *data* protected by a lock at the same time.

Acquiring a lock on an object is a matter of using the *synchronized* keyword. It also serves another purpose: You can use it to synchronize the execution of an entire method. A synchronized method effectively acquires a lock on the *this* parameter, so multiple synchronized methods of the same object instance can't execute simultaneously[42]. A quirk of the synchronized methods syntax is that overriding methods must repeat the *synchronized* keyword. It's also possible to synchronize static methods in the same way. Finally, a thread that already holds a lock can reacquire the lock again (re-enter it).

Lock granularly.

Blocking synchronization (locks) should be as granular as possible. Even though it is appealing to mark all the methods of a class as synchronized, this is rarely the recommended practice because it prevents concurrent execution of multiple threads. This is a particularly grave problem on multi-processor systems, where scalability (even in the opeating system itself) is greatly hindered by non-granular locks.

However, synchronization is rarely meant to protect only one small piece of data. Usually, synchronization is used to protect series of updates made to objects in memory, that should be observed as a single atomic (undivisable) unit.

11.4 Producer-Consumer

The producer-consumer scenario deals with a queue that mediates between a producer of work and consumers of that work. A market with customers placing

[40] This can actually be done for non-local variables using the *ThreadLocal* class.

[41] This is a highly simplified view of the problem, but it's sufficient for this discussion.

[42] Note that multiple synchronized methods of different object instances *can* execute simultaneously, and so can synchronized methods that belong to different classes. It's all a question of which lock is being acquired by the method, not the method code itself.

product offers and supplies trying to keep up with a stream of produce is a good example of this.

The shared queue must be synchronized (it must be thread-safe). However, it's not enough. For example, if at some point in time there are more customers than suppliers and the queue runs out of produce, we wouldn't want customers to sit in a tight *while* loop until there is something in the queue. Similarly, if the queue is full because there aren't enough customers, we wouldn't want the suppliers to sit in a tight loop and wait for room to become available. In both cases, we would have *busy waiting*, and this is something that must be eliminated because it's a waste of processor cycles (and also pointless energy consumption).

The proper solution for this problem is called condition variables, and it's implemented in Java as the *wait* and *notify* methods of the *Object* class. Their semantics is a little convoluted, so here's a breakdown. The *wait* method can be called iff the calling thread is holding the lock on this object, and it atomically releases the lock and transitions the calling thread into the blocked state. The calling thread remains in the blocked state until it is interrupted, until the specified timeout elapses, or until it is woken up by *notify*. The *notify* method can be called iff the calling thread is holding the lock on this object, and it wakes up one of the threads that are blocked after calling *wait*. (There's also the *notifyAll* method which wakes up all the waiting threads.) Very importantly, when the blocking thread wakes up, it waits to *reacquire* the lock—it does not immediately continue executing.

11.5 Deadlock

The problem of deadlock can be concisely defined as a group of threads waiting for themselves. The simplest deadlock is a single thread that calls *Thread.join* on its own thread instance. A slightly more complicated example is when thread A is waiting for a resource currently held by thread B, but thread B is waiting for a resource currently held by thread A. There are additional descriptions of the problem, but it all boils down to this: When the system is in a deadlock, there is no possibility of progress.

Often enough, attempting to solve deadlocks in a naive way might introduce other problems, such as livelock: A condition where threads are not blocked, but they are still making no progress because all they're doing is trying to get themselves away from a deadlock situation. Two people meeting in a narrow hallway provide an excellent illustration.

All in all, concurrent programming requires thread-safety, progress and fairness, and these are forces that pull in opposite directions.

11.6 Synchronization in the Framework

Java's framework takes multiple views on synchronization. The Swing GUI framework is not thread-safe at all because it insists that UI components should always be accessed from a single thread, the event dispatcher (EDT).

The collections framework takes a similar stand: Most collections are not thread-safe because thread-safety (locking) hinders performance in the single-threaded scenario. On top of the collection classes which are not synchronized there are special synchronization wrappers, such as *Collections.synchronizedList* which are safe for multi-threaded use.

Finally, there are some classes that are specifically designed for multi-threaded applications (most of them reside in the *java.util.concurrent* package). Among them are some collections like *BlockingQueue* which solve common concurrent programming problems. (This specific one addresses the producer-consumer scenario, including blocking when the queue is full or when the queue is empty.) Another example is the *ConcurrentHashMap* class which allows reads and writes to overlap each other.

11.7 Advanced Topics

There are Java frameworks for execution of tasks which are decoupled from specific threads. They all rely on *Executor* implementations, to which you submit a *Runnable* object to be scheduled and executed. There are some factory methods which return pre-configured execution services, including the single-threaded execution service *Executors.newSingleThreadedExecutor*, the fixed-size thread pool and other alternatives.

Futures (implementations of *Future*$<V>$) represent the result of an asynchronous method invocation, usually represented as a *Callable*$<V>$ instance. They provide the ability to enqueue a calculation and retrieve its result lazily and asynchronously when it is required.