CSC207 Course Notes (Java & Version Control)

0 Introduction to Git

0.1 What is Version Control?

Version control systems are a way for us to control the versions of our code. This means tracking the changes that we've made, and control our versions. Version control allows everyone to work on their own independent version and the version control system lets you merge your changes in as needed, keeping track of revisions as they're made.

0.2 How version control works

In version control systems, there is a master repository: a copy of the latest versions of all files. People clone the repository to get their own local copy, which they work on independently.

As people make changes and reach a state that they want reflected in the master repository, they push their changes in. Similarly, anyone who wants the latest version of the repository will pull the changes.

0.3 Git

In Git, the master repository is also known as the origin. We use the command `git clone <url>` to get a local copy of the repository. You can keep track of the files you've altered via `git status`.

0.3.1 Making changes to the origin

After modifying, adding, and removing files as needed, we may make these changes in the master repository by using `git add <files>` to add all of the files we've changed and that we want to change in the master repository.

Following the `add` command is `git commit -m "<message>"`, which we use to describe what changes we've made. Finally, to make these changes in the repository, we finish with `git push`.

Thus the general workflow for making changes is as follows:

- `git status`: Lets us see what files have changed in our local copy.
- `git add <files>`: Lets us add ("stage") files that we want to modify in the origin.
- `git commit -m "<message>"`: Saves our changes to the local repository, labelling the added changes with a message to describe our changes.
- `git push`: Pushes changes to the origin repository.

One caveat to this workflow: if there are changes in the master repository that you don't have yet in your local version, you'll need to pull these revisions.

0.3.2 Pulling new revisions

To copy over any changes into your local repository, we can use a simple `git pull` command. At this point, you may have to add/commit local changes or manually handle merge conflicts.

0.3.3 Branches

To work on an entire feature in a separate repository, it is best to make a branch which is a spin-off of the master repository, in which you can commit changes without worrying about the master branch until you're ready to merge it in.

To create a new branch and use it, you can use `git checkout -b <branch name>`. This is shorthand for `git branch <branch name>` followed by `git checkout <branch name>`: the first creates a new branch, and the 2nd switches your local repository to the new branch. The commits you make will now go `<branch name>` instead of the origin.To switch branches, use `git checkout <branch>` to change to that branch.

When you finally merge your branch into the master repository, you simply checkout your master branch, and then run `git merge <branch name>`. However, a better method is to make a pull request.

1 Introduction to Java

1.1 A first look at Java

1.1.1 Defining classes

In Java, no code is written outside of a class, and there are no functions, only methods.

1.1.2 Defining methods

In Java, there is a special method called `main` that any class may define. If we run that class the `main` method is executed. The main method must be defined with a very specific signature,
`public static void main (String[] args)`. The keyword `public` determined what code, where, is allowed to call this method.

1.1.3 Printing things

In Java, we use a method called `System.out.println` where System is a class, and out is a static data member defined in that class. It is an instance of another class that has many methods for printing things, including `println`.

In Java, we use a semi-colon to mark the end of a statement.

1.2 Variables and Types

1.2.1 Declaring Types

In Java, every value has a type, but so does every variable. We must specify a variable's type before assigning a value to the variable, and its type can never change. This is called declaring the variable. Space is reserved in memory for the variable and Java remembers to only assign it a certain type of value.

1.2.2 Flexible Python vs. Strict Java

Java makes sure we follow our type contracts, thereby avoiding many bugs.

1.2.3 Declaration and Assignment

When a variable's name is known to Java, space is reserved to store its value, and it is given a default value. For `int`s, the default value is `0`; for objects, it is `null`.

1.2.4 Keeping track of our variables

Java must keep track of four things associated with each variable:

1. The variable's name, which we provide when we declare the variable.
2. The variable's type, which we also provide when we declare the variable.
3. The memory space used to hold the value of the variable.
4. The value of the variable, which can be given with an assignment statement.

Only the value of the variable may be changed.

1.2.5 Errors

Java checks as many things as it can to avoid bugs. Some errors related to variables and types that it can detect are listed below.

1.2.5.1 Didn't declare

Java gives the error: `i cannot be resolved to a variable` when Java tried to find a variable called `i` and was unable to.

1.2.5.2 Assign value of the wrong type

Java gives the error: `Type mismatch: cannot convert from double to int` when Java tried to accommodate by converting `double` to `int` but it could not as this would cause a loss of information. However, converting from `int` to `double` is perfectly fine.

1.2.5.3 Declare a variable using a name that already exists

Java gives the error: `Duplicate local variable i` when we attempt to declare a variable of the same name more then once.

1.3 Reference Types and Primitive Types

1.3.1 More Java types

In addition to `int`, there are additional integer-valued and real-values types that allow us to either save memory (and give up precision) or gain precision (at the cost of using more memory).

1.3.2 References vs. Primitives

In Java, there are two kinds of types. Type `String` is a reference type. A Java variable cannot hold a `String` value directly inside itself; it can only hold a reference to an object of type `String`. But type `int` is a primitive type. A Java variable can hold an `int` value directly insider itself.

Primitive types all begin with a lowercase letter, and the reference types with an uppercase letter.

1.3.2.1 Primitives and References in Memory

Java keeps track of many things to run code. Every variable (its name, type, location in memory and value), every object that has been constructed, and any method that is running must be tracked.

There are three areas in memory:

- The call stack is where we keep track of the method that is currently running.
- The object space is where objects are stored.
- The static space is where static members of a class are stored.

The difference between primitive and reference types has many consequences, including when we copy a value into another variable, when we pass a parameter to a method, and when we compare two variables to see if they are equal.

1.4 Strings

1.4.1 Class String

Java has a class String that represents sequences of characters. In Java, double quotes are required for `String` literals in Java. Additionally, Java provides a short-cut for creating string objects, so you do not have to explicitly say `new`. `String` is a reference type.

1.4.1.1 String Pool

To avoid excessive memory, Java has a special "String Pool" to store the values of string literals. For example, if we create a `String` variable by `String s1 = "Hello"` without using the `new` keyword, then Java automatically throws the value of the string "Hello" into the string pool. Now if we create another string variable by `String s2 = "Hello"` without using the `new` keyword as well, to avoid excessive memory, Java goes to the string pool and looks for this phrase. Since "Hello" has already been added to the string pool, Java makes `s2` point to the same "Hello" phrase as `s1`. In this case, the expression `s1 == s2` will evaluate to `True`.

However, if two instances of the `String` class are created using `new` this will evaluate to `false` since `s1` and `s2` are different objects and the result is `false`.

1.4.2 String are Immutable

In Java, `String` objects are immutable. This means that we can never change a `String` object once it has been created. We can perform operations on `String`s, but rather than change and existing `String`, they return a new one.

1.4.3 String Operations and Methods

`String`s can be concatenated to produce a new `String` object. Class `String` also has a set of methods.

1.4.4 Mutable Strings: StringBuilder

Like a `String`, a `StringBuilder` represents a sequence of characters; however, a `StringBuilder` object is mutable. Java provides many methods for mutating a `StringBuilder`.

1.4.5 Single Character Strings: char

A `char` is a primitive type capable of holding a single character. We must use single quotes when providing a literal value of type `char`.

1.4.6 Mutating Strings vs. New Strings

We can forgo `StringBuilders` if we construct a new `String` every time we need to make a change, but constructing a new object is slower than modifying an existing one.

1.5 Classes in Java

1.5.1 Instantiating an object

In Java, we can create an instance of a class using the keyword `new`. In brackets, we provide arguments for the constructor. The class may offer more than one constructor, in which case the compiler determines which one we are calling by the number and type of the arguments.

When Java evaluates an instantiating expression, it allocates memory for the new object, evaluates the arguments, calls the appropriate constructor, and returns a reference to the newly-constructed object. We can assign the reference to a variable or use it directly.

1.5.2 APIs

The standard Java documentation specifies exactly how client code can interact with the class. The Application Programming Interface or API, tells us what methods we can call, what arguments we must send, and what value will be returned.

The API does not tell us how the class provides such services. (There are some exceptions, in cases where describing the implementation is the easiest way to describe important facts about runtime performance of the class.)

1.5.3 Calling methods

In Java, we call an instance method via an instance.

1.5.4 Class methods

"Class methods" (or "static methods", since they are defined using the keyword `static`) are associated with the class as a whole. We access a class method via the class name.

1.5.5 Accessing data members

Accessing data members (also known as attributes or instance variables) is analogous to how we access methods. If a class has an instance or class variable (also known as a `static` variables, since it is declared with the keyword `static`) that is accessible to code outside of the class, it can be referred to via an instance variable or the class name, respectively.

It is unusual to find an instance variable that we can access outside of a class, since implementation details are usually kept private.

1.5.6 Getting rid of unused objects

Java does automatic "garbage collection": it keeps track of all objects and when it can confirm that no variable refers to an object, it de-allocates that memory, making it available for other uses.

If we know that we don't need an object anymore, we can explicitly drop a reference by setting the variable holding the reference to `null`. This can hasten garbage collection and improve performance, but it may also be unnecessary and just make your code needlessly messy.

1.6 Arrays

Arrays are the simplest type that Java provides for storing a number of items. In particular:

- An array has a fixed length. It is set when we construct the array and can never change after that.
- All elements must have the same type, which we must state at the moment when we declare any variable that will refer to an array.

1.6.1 Declaring an Array

To declare an array, we must say that the type is array, which we do using square brackets, and say what type element of the array will be, which we say just ahead of the square brackets. By convention, we don't put a space between the two.

Arrays are reference types meaning that when we declare an array, we are creating a variable that will refer to an array.

1.6.2 Constructing an Array

An array in Java is a reference type, not a primitive type. As such, we must construct the object using `new`. We use the keyword `new` and the name of the type, but rather than round brackets, we use square brackets.

It is possible to to determine the size of the array when the program is running, either by reading it from the input or by computing it from other values. This is because the size of the array is not part of its type.

1.6.2.1 What values before we assign?

Each built-in type has a default value, the appropriate one is used.

1.6.2.2 Another way to construct an array: with an initializer

We can combine the contructing and initialization of an array into one step, this constructs an array of exactly the right length to hold the given values, and then assigns them to the elements of the array.

1.6.3 Determining length

We can find the length of an array by accessing its `length` attribute.

1.6.4 Accessing the array elements

Array indices start at zero in Java. Java arrays do not offer slicing and do not permit negative indices. If we try to access an array element at an index that it not between 0 and the array's length minus 1, we get a `Excepting in thread "main" java.lang.ArrayIndexOutOfBoundsException`

1.6.5 Why have such a restricted type?

Java has more flexible structures including `ArrayList`s but it takes both extra space and extra time in order to provide a list structure that can grow and shrink to arbitrary sizes. On the other other, arrays are very efficient but an array's size can never change.

1.6.5.1 Mixing types within an array is made possible by inheritance

We can use inheritance to get around the restricting that every element of array must have the same type. Every Java class is a descendent of a built-in class called `Object`. So we can simply declare that our array will hold `Object`s, and then put in any type of `Object` at all.

In Java, casting is telling Java to treat the `Object` as a specified data class, with the implied promise that when Java runs the code, this `Object` will indeed be the specified data class. If the `Object` we are casting to the specified data class does not turn out to be the specified data class, we obtain a `java.lang.ClassCastException`.

1.6.6 Two-dimensional arrays

In Java, we can create arrays with multiple dimensions.

1.6.6.1 Irregularly dimensioned arrays!

We can create arrays whose elements are themselves arrays separately, if we wish. This decoupling of the two sizes gives us the latitude to make irregularly shaped multi-dimensional arrays.

1.7 Aliases

1.7.1 Aliasing and its implications

For reference types, we must follow the reference to get to the values and methods stored in the object.

1.7.2 With references, we can create aliases

Whenever two variables reference the same object we say that they are aliases, or that we are aliasing.

1.7.3 With primitives, we cannot create aliases

In Java, if we write analogous code using primitive types instead of reference types, we do not create an alias. If we attempt to, no objects are created, but more importantly, no references are created.

1.7.4 Side-effects of aliasing

In Java, if there are two references to the same object, we must must be aware of this as changing the object that the alias variable refers to changes the object that initial variable refers to.

1.7.5 Making a copy in order to avoid side-effects

When two variables are aliases for the same object, and the object is mutable, we can have side effects. To avoid side effects, instead of making an alias, we can make a copy of the object.

1.7.6 Shallow copy vs deep copy

A shallow copy is made by copying the reference stored in the original array but not copy the objects that the original array referred to.]

Since string objects are immutable, aliasing at deeper levels does not cause significant problems. But copying an array of mutable things, has the potential to cause significant problems.

1.7.7 Other kinds of side effects

Side effects can also occur when we pass a parameter to a method.

1.8 Control Structures

In Java, curly braces {} are used to show structure.

1.9 if Statements

In Java, round brackets around the condition are required. The body itself is enclosed with curly braces. However, if the body or the if- has just a single line, the curly braces are optional. However, if more code is added without using curly brackets, Java will treat those lines as outside the if-block and will run those lines.

An if-statement can have a sequence of additional conditions denoted by `else if`, and end with an `else`. If-statements can be nested. The curly braces associate the `else` with the inner (vs the outer) if-condition.

1.9.1 for Loops

The header of a for-loop consists of three parts:

- The initialization is executes one, before any iteration begins. It often sets a counter to 0, but can be any statement.
- The termination is a boolean condition. If it evaluates to true, the loop body is first executed and then the increment is executed.
- The increment is usually a statement that increments a variable, but it could be any statement.

As long the initialization is one statement, it can be anything. The variable declaration is usually put in the initialization to limit its scope (the part of the code in which it can be referred to) to the loop. The variable disappears from the stack frame as soon as the loop is over.

1.9.2 while Loops

The structural rules of a while-loop are the same for if-statements:

1. The condition must be inside round brackets.
2. The while-loop's body needs curly braces if it is more than one line long, but it should have curly braces even if it is only one line long.

Each time the top of the loop is hit, the condition is evaluated. If the condition evaluates to true, the body is executed and goes back to the top. So when the loop terminates, the loop condition is false.

1.9.2.1 do-while Loops

The do-while loop checks its condition after the loop runs, ensuring that the loop always runs at least once.

1.10 Parameters

In the method declaration, each variable defined in the brackets is called a parameter. When a method is called, each variable in the brackets is called an argument.

When a method is called, the following happens:

1. A new stack frame is pushed onto the stack.
2. Each parameter is defined in that stack frame.

3. The value contained in each argument is assigned to its corresponding parameter. Parameter passing is essential an assignment. If an argument to a method is a variable, the value contained in the variable is assigned to the method's parameter.

Then the body of the method is executed. When the method returns, either due to hitting a `return` statement or getting to the end of the method, the stack frame for that method call is popped from the stack and all the variables defined in it - both parameters and local variables - disappear.

1.10.1 Passing a primitive

To change the value using a primitive, we must return the changed value and in the calling code, assign it to the variable we wished to change.

1.10.1.1 Passing a reference to a mutable object

Passing a reference to a mutable object, there is a potential for side effects.

1.10.1.2 Passing a reference to a immutable object

Passing a reference to immutable object, modifying the parameter has no effect outside the method.

2 Classes in Java

2.1 Classes

Classes in Java consist of attributes and methods, both private and public. It is possible to inherit from other classes, override methods, and define constructors.

2.2 Variables in classes

In Java, we have to declare variables before using them. There are two kinds of variables one can declare for classes, outside of any method:

1. Instance variables: Every instance of the class will contain its own instance of each of these variables. They come into existence when the instance is constructed (using the `new` keyword).

2. Class variables: Also known as static variables, all instances of a class share a single instance of each class variable. Updating this variable in one instance of a class will reflect across every instance of the class.

2.3 Visibility

Attributes and methods can either be public or private. In Java, we use the `public` or `private` keywords to make the distinction: private variables cannot be accessed from outside of a class. The `protected` keyword makes the attribute or method accessible to the entire package and a class' subclasses, but not to anything else.

2.4 Constructors

In Java, constructors are methods with no return type (not even void), which get called whenever a new instance of a class is created.

In Java, we can define as many constructors as we want so long as the method signatures are different: either taking a different number of arguments, having different argument types, or a combination of these. In Java, it is not required to include `this` as a parameter in our method signatures.

2.5 Overloading methods

When we define multiple constructors, we are overloading the constructor.

2.6 Overriding methods

In Java, we can override a method by re-defining the method from the parent class and using the `@Override` annotation. The annotation lets Java (and other readers) that we're are overriding an inherited method. The annotation also enforces that we use the correct method signature for the overridden method.

2.6.1 toString

One common method we will want to override is the `toString` method. This method takes no parameters and returns a `String`.

2.6.2 equals

The default behaviour of `equals` is to check for identity equality, but is often overridden to check for certain attributes.

The designer of a class decides what must be true for two instances to be considered equal. However, any implementation of it must obey these properties:

1. Symmetry: For non-null references `a` and `b`, `a.equals(b)` if and only if `b.equals(a)`.
2. Reflectivity: `a.equals(a)`
3. Transitivity: If `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`

2.6.3 hashCode

The hashCode of an object is an integer value that obey that if two objects are equal (according to the equals method), they have the same hashCode. Two objects with the same hashCode may not be equal.

2.7 Class (static) methods

The keyword `static` is used to declare that a method is a class method. Since a class method is associated with the class and not the instance, it is called by prefixing it with the class name.

An instance method can reference a class variable (or call a class method), but a class method cannot access an instance variable or call an instance method directly. There is no "this" in a class method. The only way for a class method to access an instance variable or call an instance method is if a reference to an object is passed to the method. Through that reference, the instance variables and instance methods of the object are accessible.

2.8 Comparable

2.8.1 Being comparable enables sorting

Java provides a `sort` method capable of sorting an array of `ints` or any of any other primitive type. Note, `sort` is overloaded: there is a series of `sort` methods, each one capable of handling of the primitive types. They are all defined as static methods in the `Arrays` class.

The `sort` method requires that all elements in the array must implement the `Comparable` interface. The interface in turn requires any class that implements it to define the `compareTo` method which compares two objects and returns a negative integer, zero, or positive integer. The `sort` method also requires all elements in the array must be mutually comparable (unless the classes which the objects are instances of share a parent class implementing `Comparable`.

2.8.2 Being comparable enables comparisons

We can compare built-in classes, such as `String`, `Integer`, `Double`.

2.8.3 Making our own classes Comparable

When the requirements of implementing the `Comparable` interface are fulfilled, we must change the class declaration to `class Something implements Comparable<Something>{}`. Note `Comparable<Something>` promises that an instance of this class can be compared to any other instance of `Something`. If the class implements `Comparable`, the `compareTo` must accept and deal with any `Object`

2.8.4 Being comparable enables more

In addition to enabling sorting, a class that implements `Comparable` can be used in certain ”Collections” that care about order.

2.9 Comparator

A ”comperator” class implements the `Comparator` interface, which requires the `compare` method which compares two arguments for order.

2.9.1 When to use Comparable vs Comparator?

If you are not the author of the class, you cannot make it `Comparable`. Your only option is to define one or more comparators. If you are the author of the class, both options are available to you.

3 Relationships between Classes

3.1 Inheritance

In Java, inheritance uses the `extends` keyword is used instead of brackets. The child class inherits all of the methods and variables defined in the parent class, and the child class is an instance of the parent class.

3.1.1 Abstract classes

In Java, the `abstract` keyword is used to signify that a class is abstract, this enforces that no instance of the class should be created, even if there are no abstract methods in the class. The `abstract` keyword can be used for any abstract methods. Any non-abstract class that extends an abstract class has to implement the body of all abstract methods.

3.1.2 Overriding methods

In Java, we override a parent class' methods by redefining it and including an `@Override` annotation. This informs the compiler that method is meant to override an element in a superclass. Although the annotation is not required, including it helps prevent errors.

3.2 Interfaces

In Java, we can only extend a single class: we have one parent class. To define a property of a class, we use interfaces. Interfaces are similar to classes, except they have no implementation details: only method signatures. They can also have variables, but these variables must be `static` and `final`. In addition, everything in an interface must be `public`. We can implement as many interfaces we want. Interfaces can `extend` other interfaces (not `implements` – an interface doesn't implement anything.)

3.3 super

In Java, the `super` keyword is used to refer to methods in the parent class. To call a parent's constructor, we use `super()`, or `super(a,b,c)` if parameters must be passed. To call a parent's method, we use `super.method()`.

3.3.1 Constructors with super

To extend another class, Java requires a call to the superclass' constructor to be made in the subclass' constructor. Furthermore, this call must be the first thing done. If no constructor call is explicitly made in the subclass' constructor, then an implicit call to `super()` is made. It is best to explicitly include `super(...)` calls in constructors to precisely know which constructor is being called by the subclass.

3.4 Polymorphism

Polymorphism is the ability of an object to take many forms. An object is considered polymorphic if it passes multiple `instanceof` tests.

An example of a polymorphsim in use is when we have a variable whose value may be of a type other than the variable's type itself.

3.5 Casting

Casting is when we change the type of an object to another, often in order to access more specific functionality. Downcasting is when the type of a variable is cast into its subclass. Upcasting is when the type of the variable is cast into its superclass.

3.5.1 Primitive conversions

Some primitives can be cast, such as converting an `int` into a `double` and vice versa. Although, casting is a re-labelling of the type, when casting primitive the value of the variable itself is adjusted and the changes made are potentially irreversible.

4 Assorted Topics in Java

4.1 Shadowing

Variable shadowing occurs when the same variable is used in two different scopes. We use `this` to refer to the class variable, while no indicator is used for the local variable.

4.2 Array Copy

In Java, using `clone()` creates a copy of the outermost arrays, but not copies of inner arrays. To make a deeper copy.

4.3 Autoboxing

Autoboxing is a conversion that the Java compiler makes automatically between primitive types and their corresponding object wrapper class and vice versa.

4.4 Generics

Generics are a way for programmers to generalize the type that a class works with. It allows programmers to re-use the same code but allow for various input types without the need to cast things constantly.

4.5 Collections

A collection is an object that represents a group of objects. Java defined an interface called Collection that defined the operations that any collection should offer, including operations to add and remove items, and to find out the number of items in the collection.

The interface is very general. For instance, it doesn't specify the order in which items are removed or whether duplicates are allowed. Java defines more specific interfaces that specify these things more fully and add more methods. These include interfaces `Queue`, `List`, and `Set`.

4.5.1 Implementations

Each interface has several different implementations, with different characteristics.

4.5.2 The Java Collections Framework

The various interfaces, their implementations, provided static methods that operate on collections, and some additional infrastructure are together referred to as the "Java Collections Framework".

4.5.2.1 List

Java's List grow and shrink as needed. `ArrayList` is one implementation that allows quick access to elements by index. To get primitives into an `ArrayList`, we must use a wrapper class.

All elements of a `List` must be objectives, not primitives, and they must be of the same type. However, we can use inheritance to allow mixed types.

4.5.2.2 Set

A `Set` does not allow duplicate elements and has no notion of elements being in any particular position. `TreeSet` is one implementation. `TreeSet` has a `toString` method and implements the `Iterable` interface which provides a `hasNext` and a `next` method.

4.5.2.3 Map

The `Map` interface maps key to values, and each key can have only one value associated with it. `Keys` do not have to immutable. However, we must know what we are doing if we mutate an object after it has been added as a key, otherwise, the behaviour of `Map` will be undefined. `HashMap` is one implementation of `Map` using a hash table.

`HashMap` offers constant-time performance for the basic operations (`get` and `put`). However, this depends upon certain properties of the hash table being maintained as key-value pairs are added and removed. When we construct `HashMap`, we can control parameters of the data structure that may help ensure these properties are maintained.

5 Exceptions in Java

5.1 What are Exceptions?

Exceptions report exceptional conditions: unusual, strange, unexpected. These conditions deserve exceptional treatment: not the usual go-to-the-next-step, plod-onwards approach. Therefore, understanding exceptions requires thinking about a different model of program execution.

5.2 Exceptions in Java

To throw an exception, we use `throw Throwable();`.

To catch an exception and deal with it, we use

```
try {
    statments
} catch (Throwable parameter) {
    statements
}
```

To say a method isn't going to deal with exceptions (or may throw its own), we say

```
accessModifier returnType methodname (parameters) throws Throwable {
...}
```

.

5.3 Why use Exceptions?

Less programmer time is spent on handling errors. We obtain a cleaner program structure by isolating exceptional situations rather than sprinkling them throughout the code. Concerns are separated by paying local attention to the algorithm being implemented and global attention to errors that are raised.

5.4 We can have cascading catches

Much like an `if` with a series of `else if` clauses, a `try` can have a series of `catch` clauses. After the last `catch` clause, you can have a `finally {...}`. But `finally` is not like a last `else` on an `if` statement. The `finally` clause is always executed, whether an exception was thrown or not, and whether or not the thrown exception was caught.

5.5 Summary

If you call code that may throw an exception, you have two choices, you can warp the code in a `try-catch`, or you can use `throws` to declare that your code may throw an exception. Exceptions don't follow the normal control flow. Exceptions should be used for exceptional circumstances. Throwing and catching should not be in the same method.

5.6 You don't have to handle Errors or RuntimeExceptions

Errors indicate serious problems that a reasonable application should not try to catch. We do not have to handle these errors because they are abnormal conditions that should never occur.

`RuntimeException` are called unchecked because you do not have to handle them. This is a good thing, because so many methods throw them it would be cumbersome to check them all.

5.7 Some things not to catch

Don't catch `Error`: You can can't be expected to handle these. Don't catch `Throwable` or `Exception`: Catch something more specific.

5.8 What should you throw?

You can throw an instance of `Throwable` or any subclass of it (whether an already defined subclass, or a subclass you define). Don't throw an instance of `Error` or any subclass of it: These are for unrecoverable circumstances. Don't throw an instance of `Exception`: Throw something more specific. It's okay to throw instances of specific subclasses `Exception` that are already defined and specific subclasses of `Exception` that you define.

5.9 Don't use Throwable or Error

`Throwable` itself, and `Error` and its descendants are probably not suitable for sub classing in an ordinary program. `Throwable` isn't specific enough. `Error` and its descendants describe serious, unrecoverable conditions.

5.10 Aside: Classes inside other classes

You can define a class inside another class. There are two kinds.

Static nested classes use keyword `static` (It can only be used with classes that are nested.) You cannot access any other members of the enclosing class.

Inner classes do not use keyword `static`. These can access all members of the enclosing class (even private ones).

Nested classes increase encapsulation. They make sense if you won't need to use the class outside its enclosing class.

5.11 What does the Java API say?

Class `Exception` and its subclasses are a form of `Throwable` that indicate conditions that a reasonable application might want to catch.

`RuntimeException` (not checked) is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

`non-RuntimeException` are checked.

5.12 What does the Java language specification say?

The runtime exception classes (`RuntimeExcpetion` and its subclasses) are exempted from compile-time checking because, in the judgement of the designers of the Java programming language, having to declare such exceptions would not aid significantly in establishing the correctness of programs.

The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared would simply be an irration to programmers.