Hands-On Java

Getting Started

File System Elements and Program Entry

Packages:

- Group related code
- Are implemented as directories

Source files:

- Have the extension `.java`
- Contain classes (and other types; abstract classes, records, and interfaces)
- Name must match any `public` type in the source file

The program entry point:

- Is a method that must be declared exactly as:

```
public static void main(String [] args) { }
```

Imports

Packages and imports:

- When using a feature of a library the fully qualified name includes the package name, the classname, and the element name (e.g. method or value)
- An import can bring the name into the local scope, allowing it to be used unqualified

Java supports two kinds of import:

- Regular imports bring a named type, or all the types in a package into scope

  ```
  import java.util.List;      import java.util.*;
  ```

- In this case, elements of the list can be used with a qualifying prefix

  ```
  List.of(a,b,c);
  ```

- A static import allows a static member of a type to be accessed with no prefix

  ```
  import static java.util.List.of;   of(a,b,c);
  ```

The types in the package java.lang are all implicitly imported

- but static members must be imported to be used without prefix

  ```
  import static java.lang.System.out;  out.println("Hello");
  ```

Console Output

Text can be sent to the command-line / console using methods of the out object in the `System` class:

```
System.out.println("Hello World!");
```

The `java.lang.String` class supports concatenation with the plus operator

```
System.out.println("Hello" + " World!");
```

- Any time `+` is used and either of the operands is a `String`, the other will be converted to `String`
- This conversion is possible for all of Java's types, including programmer-defined types, although the result might be less than helpful in some cases

The `System.out` object provides a `printf` method that supports formatted printing

- Check documentation for the `Formatter` class for details

Command-line Basics

Java can be compiled and run using only the command line. This is frequently automated in a build system.

To run the compiler, we must tell it what files we want compiled, where to find supporting binaries, and where to put the compiled output files. The javac command invokes the compiler, and the basic three elements are provided like this:

```
javac -d <output-dir> -cp <path-to-binaries> <files-to-compile>
```

To run the generated binary, we must tell the system what class has our program entry point, and where to find the binaries. The java command launches the JVM, and the elements are specified like this:

```
java -cp <path-to-binaries> fully.qualified.ClassContainingMainMethod
```

Variables, Expressions, Primitives and References

Variables, Declarations, and Initializations

Java literals have unambiguous type:

```
int     - 1234, 1_234
long    - 1234L
double  - 1234.0, 1234D, 1234E6, and combinations
float   - 1234F, and combinations
String  - "Hello\n", double quotes, escape sequences for some special characters
String  - """
          Hello, text blocks"""
```

Java requires all variables to be declared with an unambiguous type. The var pseudo-type infers the type from the required assignment, it does not create dynamic typing

```
int count;   String name = "Fred";   var pi = 3.14159
```

Java requires all local variables to be definitely initialized before reading

- This is sometimes surprising if initialization is attempted in a conditional, or try, block

Assignment Compatibility

Java allows assignments from the same type, from "narrower" types, and from int literals that "fit" the target of the assignment:

These work:

```
boolean bool = true; // boolean literals are true and false
byte b = 99; // int literal 99 fits in a byte
long lng = 2_000_000; // int widens to long
float fPi = 3.14F;
double pi = fPi; // another widening conversion
int count = 10;
```

These fail:

```
b = count; // int cannot generally be assigned to a byte
float f = 3.14; // can only assign int literals to narrower types
```

Java does not automatically convert anything to `boolean`, there is no concept of "truthy and falsy"

Using Casts

If the programmer knows that a narrowing assignment is safe, they can use a "cast" to force the compiler to permit the assignment.

However, if the assigned value cannot be represented, the result is garbage:

```
int x = 100
byte b = (byte)x; // b contains 100 after this

x = 420;
byte b = (byte)x; // b contains -92 after this!
```

The effect of the second assignment is that the low-order 8 bits of the value 420 (which happen to be 10100100) are assigned to the byte, resulting in a rather meaningless value.

Casts have additional meaning / effect with reference types and lambda expressions.

Key Operators

Java provides arithmetic, comparison, and logical operators

`+`	Numeric addition and String concatenation	`==`	Expression value equality
`-`	Subtraction	`!=`	Expression value inequality
`*`	Multiplication	`<`	Less than
`/`	Division	`<=`	Less than or equal
`%`	Remainder (not mod*)	`>`	Greater than
`+, -`	Unary positive and negative	`>=`	Greater than or equal
`++, --`	Increment and decrement	`&&`	Conditional and
`!`	Logical negation	`||`	Conditional or

*Note that remainder and mod have different results if negative values are involved

Key Operators

Java provides bit manipulation operators and a type classification operator

`~`	Unary bitwise complement	`instanceof`	Determines if an object is assignable to the specified type
`<<`	Signed left shift		
`>>`	Signed right shift		
`>>>`	Unsigned right shift		
`&`	bitwise and		
`^`	bitwise exclusive or		
`||`	bitwise or		

Java does not permit user-defined operator overloading

Primitive and Reference Types, and Equality Comparison

Java has two distinct expression types, primitive and reference

There are 8 primitive types, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`, and in Java 17, *any other type* is a *reference type*

- The value of an expression of primitive type is what we think of as the value
- The value of an expression of reference type is similar to a pointer, it tells where to find the data, rather than actually being the data
- This is critically significant when using `==` which compares the expression values, therefore `==` used on any reference expression indicates if the two expressions are aliases for the same data in memory
- To compare "content-equivalence" the `equals` method is useful in *some* (but not all) data types. (When not implemented usefully, it exists, but degenerates to `==`)
- A reference expression might have a special value indicating that there is no data, this is the value `null`

Flow Control

The `if` Statement

Java has an `if` / `if` - `else` statement

- The `if` keyword is followed by a boolean test expression in parentheses
- An immediately following statement or block is executed if the test evaluates to true
- An optional `else` takes another statement or block
- No elif is needed, simply use `else if` ...

```
if (someTest()) {
  out.println("it's true!");
} else if (someOtherTest()) {
  out.println("the second one is true");
} else {
  out.println("neither were true");
}
```

The Conditional Operator

Java has a conditional operator

- A `boolean` test expression is enclosed in parentheses and followed by a question mark
- An expression follows which is the value of the of the overall expression if the test is true
- Next comes a colon and another expression, this second expression is the value if the test was false

```
var message = someTest() ? "test was true" : "test was false";
```

The type of the result is subject to quite complex rules, but is generally a type that can describe either of the expressions, and might degenerate to `Object` if they are very different.

The `switch`/`case` Statement (Old Style)

Where multiple possible matches might exist, the `switch/case` statement might be useful:

```
switch (matchThis) {
  case 0:
    out.println("It's zero");
    break;
  case 1: case 2: case 3:
    out.println("1, 2, or 3");
  case 4: case 5:
    out.println("1, 2, 3, 4, or 5!");
    break;
  default:
    out.println("It's something else");
}
```

Note that used in this way, cases "fall through" if no `break` is found.

The `switch`/`case` Statement (Java 14+)

Fall-through in the absence of `break` has caused bugs in the past, the new style changes this:

```
switch (matchThis) {
  case 0 -> {
        out.println("It's zero");
        out.println("nothing at all!");
      }
  case 1, 2, 3 ->
    out.println("1, 2, or 3");
  case 4, 5 ->
    out.println("only 4 or 5!");
  default ->
    out.println("It's something else");
}
```

In this version, a single statement, or a curly-brace block, follows each arrow.

The `switch`/`case` Expressions

Another addition at Java 14 is the ability to use the `switch` as an expression:

```
var message = switch (matchThis) {
  case 0 -> "It's zero";
  case 1, 2, 3 -> "1, 2, or 3";
  case 4, 5 -> "only 4 or 5!";
  default -> "It's something else";
};
```

Either the colon or arrow syntax versions can become an expression. This happens by putting the `switch`/`case` structure in a place where an expression is needed. In this situation, a single statement, or a curly-brace block, follows each arrow and all possible values of the switch variable must be accounted for.

The `while` Loop

Java provides a `while` loop:

```
int count = 3;
while (count > 0) {
  out.println("counter is " + count);
  count--;
}
out.println("Done counting");
```

This loop continues execution as long as the test is true. It might execute zero times if the test is false before the loop ever runs.

The `do`/`while` Loop

Java also provides a `do` / `while` loop:

```
int count = 0;
do {
  out.println("counter is " + count);
  count--;
} while (count > 0);
out.println("Done counting");
```

This loop structure requires a semicolon after the test.

As with all `do`/`while` constructs, this always executes the body at least once.

The C-style `for` Loop

Java provides a `for` loop modelled on the C/C++ programming languages:

```
for (int count = 3; count > 0; count--) {
  out.println("counter is " + count);
}
out.println("Done counting");
```

The three elements in parentheses are:

1) Initialization, which can declare and initialize zero or more variables of the same base type, or contain zero or more "expression statements". Note that variables declared here go out of scope at the end of the body of the `for` loop
2) Test, which must be a boolean expression and behaves the same as the test in a `while`
3) Update, which allows zero or more "expression statements", and is usually used to alter the controlling variable so that the loop eventually terminates.

Expression Statements

Expression statements include assignments, increments, decrements, method invocations, and constructor calls. These can be used in `for` loops to create code that's either succinct or simply very difficult to read:

```
int count = 3;
for (out.println(count), new Object(); // new Object is pointless!!
     count > 0;
     count++, count--, count--, out.println("silly increment!")) {
  out.println("counter is " + count);
}
out.println("Done counting");
```

Syntax permits *either* variable declaration/initialization *or* expression statements in the initialization part of this loop. (They cannot be mixed)

Multiple expression statements in a `for` construction are separated by commas.

The Enhanced `for` Loop

Java provides a `for` loop for extracting each item in turn from an array or collection:

```
for (String s : manyStrings) {
  out.println("String is " + s);
}
```

This loop does not keep track of the item number of the elements pulled from the data set

This loop is usually almost always more efficient than an indexing approach for accessing a `List`

Reading Text

Using a `Scanner`

Java provides `System.in` (compare with `System.out`) as a connection to the standard input channel of the JVM. This is hard to use by itself, so it's usual to wrap a utility like a `Scanner` around it.

```
Scanner sc = new Scanner(System.in);
while (sc.hasNextLine()) {
  String line = sc.nextLine();
  out.println("Read: " + line);
}
out.println("all done");
```

Reading a File

A `Scanner` is not restricted to the standard input channel, it can read from any `InputStream` or `Reader`, but file I/O generally requires the programmer to consider how to behave if no file is found or other IO problems arise. These problems are reported using exceptions that must be addressed. This example does not show any mandatory exception handling.

```
Scanner s = new Scanner(Files.newBufferedReader(Path.of("myfile.txt")));
while (s.hasNextLine()) {
  String line = s.nextLine();
  out.println("Read: " + line);
}
out.println("all done");
```

Reading a File

A `Scanner` can also read directly from a file. This example expressly ignores the exception by declaring the method throws it. This is sufficient to ensure the code compiles, but it will crash if the file is not found.

```
public static void main(String[] args) throws IOException {
  Scanner sc = new Scanner(Path.of("myfile.txt"));
  while (sc.hasNextLine()) {
    String line = sc.nextLine();
    out.println("Read: " + line);
  }
  out.println("all done");
}
```

The factory `Path.of` identifies the file to be read. Prior to Java 11, use `Paths.get` instead.

Introduction to Arrays

Array Overview

Java arrays have some significant features:

- The element / base type is known to the compiler, and is checked on access by the runtime
- The length of an array is fixed at the moment it is created; arrays neither grow nor shrink
- Multi-dimensional arrays are simulated using arrays of arrays, this means they do not have to be rectangular

Arrays declarations take the name of the element / base type followed by pairs of square brackets, one pair for each dimension to be created. The declaration, and the variable declared, does not know the dimension(s) of the array. The variable name is conventionally placed after the square brackets.

```
String [] names;
int [][] coordinates;
LocalDate [][][] cubeOfDates;
```

Initializing Arrays

Arrays can be initialized programmatically, or using literals.

```
String [] names = new String[3];
names[0] = "Fred";

int [][] coordinates = {{10, -2},{7, 3}};
int [] values = new int[]{9, 22, 3, 17, 25, 7};
```

Note the coordinates variable uses *type inference* for the array type on the right hand side. The values array uses a type-explicit construction. In both of those examples, the size of the array is inferred from the initializing values.

Accessing Arrays

Array elements are accessed using the square bracket subscript form and a zero-based index

```
int [] values = new int[]{9, 22, 3, 17, 25, 7};
out.println("First value is " + values[0]);
```

Java arrays are always contiguous, and indexed from zero to length - 1.

```
out.println("Last value is " + values[values.length - 1])
```

Iterating arrays is easily performed using the enhanced for loop

```
int[] values = new int[]{9, 22, 3, 17, 25, 7};
int sum = 0;
for (int v : values) {
  sum += v;
}
out.println("Total is " + sum);
```

Introduction to Functions

Static Methods

Java requires methods to be declared inside types (commonly classes, but also interfaces and records). A method cannot be declared at the top level of a source file.

A method with the modifier `static` is called a static method (or sometimes a function).

Many static methods are declared with this general template:

<accessibility> `static` <return type> <method name> `(`<formal parameters>`) {`<statements>`}`

Examples:

```
public static void showMessage(String msg) {
  out.println("The message is " + msg);
}

public static int add(int a, int b) {
  return a + b;
}
```

Rules of Static Method Declarations

Every method must be declared with parentheses to enclose the formal parameter list, but the formal parameter list may be empty.

Formal parameters consist of a type-name followed by a local variable name.

Expressions in the caller are passed to formal parameters by value, however, the value of any expression of reference type is a reference. This often causes confusion; many believe that objects are "pass by reference" but this is inaccurate.

A method must declare a return type, if no value is to be returned, mark the return type as `void`.

For any return type other than `void` every path out of the method, other than throwing an exception, must explicitly return a value of an assignment compatible type.

Overloading Method Declarations

The full name of a method includes the package name, the classname, the base method name, and the type-sequence of the arguments. Consequently:

- Methods in different classes can have identical names and type sequences
- Methods in the same class can have identical base names if they take a different type-sequence for their arguments
    - This is called "method overloading"
    - Conceptually, method overloading allows us to describe ways to "do the same thing with different raw materials"--like baking a cake with flour, eggs, and milk, or baking a cake from a packet mixture.

It's important to know that the return type of a method is irrelevant for the purpose of identifying it.

Some superficially-different argument types don't actually qualify as different for this purpose, notably variable length argument lists and arrays, and generic types such as `List<String>` and `List<LocalDate>`, which are indistinguishable at runtime.

Invoking Methods

Given a static method has been declared, it can be invoked by giving the fully qualified classname, a dot, the method name, and a comma separated list of expressions of types that are appropriate to the formal parameters.

If the method declares a return type, the invocation is itself an expression of that type, and can be assigned to a variable or used in any way valid for an expression.

If the class containing the method is imported or is in the same package as the invocation, the package prefix can be omitted, and if the invocation is in the same class, the classname can be omitted too.

Examples of Invoking Methods

Assuming this method declaration:

```
package arithmetic;
class Summer {
  public static int add(int a, int b) { return a + b; }
}
```

These are all, individually, potentially valid invocations:

- `int result = arithmetic.Summer.add(1, 3);`
- `int result = Summer.add(1, 3);`
- `int result = add(1, 3);`
- `Summer.add(1, 3);`
- `int result = Summer.add(Summer.add(1, a * b), 99);`

Invoking Overloaded Methods

Assuming these method declarations (which are all valid as shown):

```
class Summer {
  public static int add(int a, int b) { return a + b; }   // method 1
  public static long add(int a, long b) { return a + b; } // method 2
  public static long add(long a, int b) { return a + b; } // method 3
}
```

- `add(1, 2)` invokes method 1
- `add(1, 2L)` invokes method 2
- `add(1L, 2)` invokes method 3
- but, if method 1 were removed, a *call* to `add(1, 2)` would be *disallowed*. The compiler could not decide between promoting 1 to long and invoking method 2, or promoting 2 to long and invoking method 3.

Notice that it's ambiguity at the *call site*, rather than the declarations that cause trouble here.

An Array in the Argument List

If a method should be able to process a variable number of elements of the same type, these can be passed as an array, like this:

```
public static int add(int [] values) {
  int sum = 0;
  for (int v : values) sum += v;
  return sum;
}
```

In which case, the method can be called like this:

```
int total = add(new int[]{9, 22, 3, 17, 25, 7});
```

Variable Length Argument Lists

A method also receives a formal parameter as an array if that formal parameter is declared using the ellipsis format like this:

```
public static int add(int ... values) {
  int sum = 0;
  for (int v : values) sum += v;
  return sum;
}
```

In which case, the method can be called like this:

```
int total = add(9, 22, 3, 17, 25, 7);
```

At most one "varargs" parameter can exist in a method declaration, and it must be the last formal parameter of that method.

Note that this mechanism is purely a syntactic convenience, it only results in cleaner source code in the caller than if a regular array had been used.

Exceptions

Throwing an Exception

Java provides exceptions as a means to indicate that a situation has arisen that requires altering the normal program flow.

- At the point where a problem arises, code can "throw" an exception
- The exception itself is an object (some subclass of `Throwable`) that describes the problem
- Execution flow jumps to a handler (if one exists); a handler is provided in a `catch` block that is written for that type of exception object, or a parent class of that type
- A `catch` block must be directly associated with a `try` block

Java categorizes exceptions into "checked" and "unchecked". Conceptually, unchecked exceptions were supposed to represent either program bugs (which should not be recovered from, since the state of the program cannot be known) or catastrophic failures from which recovery is impractical. Checked exceptions were intended to represent those situations (for example, a file not being found) where the program is in a sound state and therefore some recovery action is both sensible and desirable for good user experience.

Handling Checked Exceptions

If the Java compiler determines that a checked exception might arise at some point in a method, one of two actions are required in the source code:

- Surround the code containing the potential exception source with a `try` block that carries an associated `catch` block that will be invoked in response to the problem
- Declare that the method `throws` that exception or a parent of it

Because any unhandled checked exception arising from a method must be declared, the same rules are imposed on the caller of such a method. Consequently, the compiler can effectively force the programmer to "do something", rather than forget about the problem or assume "that won't happen".

Note that popular usage commonly replaces checked exceptions with unchecked ones, allowing programmers to simply ignore the potential problems.

Exception Handling Code - 1

The basic structure of `try`/`catch` looks like this:

```
public static int mightBreak(int x) {
  try {
    out.println("x is " + x);
    if (x > (int)(Math.random() * 100)) {
      throw new SQLException("Pretending the DB broke");
    }
  } catch (SQLException sqle) {
    out.println("trying to recover from DB problem");
  }
  return x;
}
```

Exception Handling Code - 2

It's valid to catch a parent class of the exception that's thrown:

```
try {
  out.println("x is " + x);
  if (x > (int)(Math.random() * 100)) {
    throw new SQLException("Pretending DB broke");
  }
} catch (Exception sqle) {
  out.println("trying to recover from DB or other problem");
}
```

Exception Handling Code - 3

Structurally, multiple catches are permitted:

```
try {
  // code that might throw an SQLException
  // or might throw FileNotFoundException
} catch (SQLException sqle) {
  out.println("trying to recover from DB or other problem");
} catch (FileNotFoundException fnfe) {
  out.println("File was not found");
}
```

Note that a `catch` block for a checked exception that is known to be impossible is rejected

Exception Handling Code - 4

A common ancestor class can be caught to handle multiple exception types:

```
try {
  // code that might throw an SQLException
  // or might throw FileNotFoundException
} catch (Exception ex) {
  out.println("Something bad happened");
}
```

Note that this catch block will catch ***any and all*** subclasses of exception including all the exceptions that represent bugs. *This is likely not what's intended*.

Exception Handling Code - 5

Where it's desirable to use common code to handle multiple exception types a better approach is to use the multi-catch form:

```
try {
  // code that might throw an SQLException
  // or might throw FileNotFoundException
} catch (SQLException | FileNotFoundException ex) {
  out.println("Either the DB broke, or the file wasn't found");
}
```

In this form only the specifically listed exceptions will be caught.

The `finally` Block

Java also provides a `finally` block:

```
try { // code that might throw an Exception
} catch (FileNotFoundException | SQLException ex) {
  out.println("DB or file problem");
} finally {
  out.println("This always executes");
}
```

If the `try` block starts executing, the `finally` block will definitely start executing after:

- the `try` block completes normally
- or a `catch` block completes normally
- or an exception arises that is not handled

A `finally` block might not complete if an exception arises (or the VM shuts down) during execution of the `finally` block, or if shutdown occurs before the `finally` can start.

Closing Resources

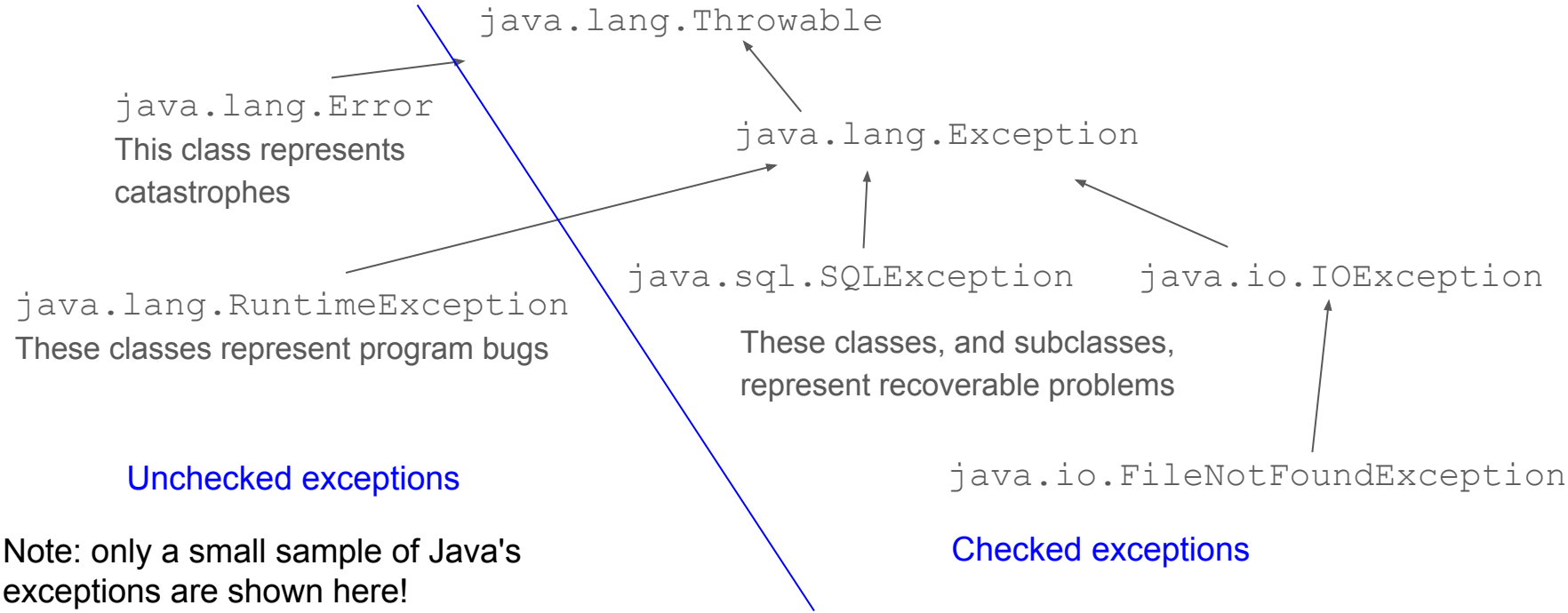
The `finally` block was intended to provide for reliable release of resources such as file handles, however, it's cumbersome to use properly. Java now has the "`try` with resources" structure:

```
try ( // NOTE this is a parenthesis, not a curly brace
    FileReader fr = new FileReader("data.txt");
    Scanner sc = new Scanner(fr);
) {
  out.println(sc.nextLine());
} catch (IOException ex) {
  out.println("DB or file problem");
}
```

The compiler builds a `finally` block to close these resources (`fr` and `sc` in this example) properly and reliably, in reverse order from their opening. Resources must implement the interface `AutoCloseable`. A `finally` block can be added too, which would run after the closure of the resources. This is not usually useful, however.

Exception Hierarchy

Java has many exceptions, and programmers can define their own too. It's useful to understand the broad categories and the classes that capture them:



Note: only a small sample of Java's exceptions are shown here!

Re-Throwing Exceptions

How a piece of code can fail, and how it reports that failure, is part of its public interface. This means that if these details change, client code is likely to need to be rewritten to accommodate the change.

To minimize the risk of this, it might be a good idea to report errors in terms meaningful to the current abstraction, rather than implementation detail. To this end, it might be appropriate to create a custom exception to describe the problem.

However, if the problem first shows up as an implementation specific exception, this should not be lost. Java allows one exception to be embedded in another, so that the "cause" is not lost.

Declaring a Custom Exception

Any exception declared by the programmer must be a subclass of `Throwable`, but subclassing `Exception` is more appropriate. If the exception should be unchecked, subclass `RuntimeException` instead. It's a good choice to provide all the constructor variants suggested by the `Exception` class.

```
public class CreditDeclinedException extends Exception {
  public CreditDeclinedException(String message, Throwable cause) {
    super(message, cause);
  }
  // several other constructors that delegate to
  // the superclass should also be added
}
```

Throwing a Custom Exception

If a custom exception has been defined as shown previously, it can be thrown like this:

```
public static void spendMoney(int money)
    throws CreditDeclinedException {
  try {
    // operation that might fail with implementation exception
    if (Math.random() > 0.5)
      throw new ArithmeticException("bad number");
  } catch (ArithmeticException ae) {
    // report to caller as CreditDeclined...
    throw new CreditDeclinedException("Not enough money", ae);
  }
}
```

Introduction to Interfaces and Generics

Interface vs Implementation

Often there are multiple ways to implement a concept. For example, a list can use an array or a linked structure. The interactions are the same, but details such as performance implications can be entirely different. To support this, Java provides interfaces. An `interface` allows specification of method signatures–describing how to *interact* with something–without specifying how it might be implemented. (Some other languages do this with pure abstract classes.)

Java interfaces support:

- specification of public method signatures without implementations (abstract methods)
- implementation of concrete static methods, either public or private
- private instance methods
- "default" instance methods, which provide a fallback implementation in case the target class does not. These are often used for adding methods to popular library interfaces

Classes can "implement" *multiple interfaces*, but class/implementation inheritance is restricted to a *single parent class* in Java

Interactions with Lists

Both `LinkedList` and `ArrayList` implement the `List` interface. Key features they provide include:

- A user-ordered sequence of items
- Automatic resizing of storage when needed for newly added items
- Ability to set or get an item at a particular index position (with zero based indexing)
- Ability to insert an item at a particular index, shifting other items right to make space

Notably, the Java's list structures can contain *any reference type*. Primitives types can be added only if they are boxed into their wrapper type.

Introducing Java's Generics Mechanism

The behaviors of `List` implementations are coded in terms of `Object`, allowing the list to contain any reference data type. This is useful, but in isolation allows the wrong data type to be inserted into a list by mistake, and means that what's removed is of unknown type.

The generics mechanism allows the programmer to tell the compiler the intended content type for any particular list instance, and in that way to improve the readability and correctness of the code.

Note that the generics system is broadly applicable, and is not limited to use with lists.

Using Lists With Generics

If a list is used *with* the benefit of generics, the programmer declares the intended contents using the angle-bracket syntax, and the compiler validates uses of that list. The compiler also adds the cast to the generated code when reading from the list, simplifying the source code.

```
List<String> stuff = new ArrayList<>();
stuff.add("Hello");
stuff.add(99); // Compiler rejects this
String st = stuff.get(0); // compilation succeeds, cast not needed
```

The syntax `List<String> stuff` indicates that when using `stuff`, the list will contain Strings.

The syntax new `ArrayList<>()` tells the compiler to guess the content type of the expression from the context. Context would include the argument list, if there were one, and the target of the assignment. Of course, the compiler determines that it's `String` here too. Note that `new ArrayList<String>()` is also valid syntax, though it's more verbose.

A Limitation of Generics

The generics mechanism is very powerful and lightweight. However, it's implemented entirely as a "consistency of use" checking system by the compiler. The generated code is not changed (this is what makes it lightweight, of course). As a result, a list itself does not actually know what type it's supposed to contain. As programmers, we simply trust that the compiler will tell us if we do something that is not provably safe.

However, the generics mechanism did not exist prior to Java 1.5, so older code might exist in a large codebase. If we mix "legacy" code with newer, we might find the wrong data types sometimes added to our lists. Use generics consistently, and update any old code you find, and this should not be a significant problem, however.

This mechanism is known as "type erasure" since the generic type information is erased after the compilation completes. There are some other, more esoteric, consequences of this which are beyond our scope.

More of Java's Collection API

Java provides more data structures than just lists. Each has an interface and might have several alternative implementations. Key interfaces are:

- `Collection` – this describes "many things" (similar to a list), but there is no particular order. A collection might permit duplicates, or not, and might allow null items, or not. `Collection` is the parent interface of both `List` and `Set`
- `Iterable` – this describes "many things", similar to a `Collection`, but the items might not be stored (so, they might vanish once taken from the `Iterable`). The main feature of an `Iterable` is that the programmer can inquire whether there is another item available, and if so, can request that item. The enhanced `for` loop can be used directly on any object that implements `Iterable`. `Iterable` is the parent interface of `Collection`.
- `List` – already described, but this is a collection that maintains a user defined order, and might or might not permit duplicates, and or nulls
- `Set` – this is another type of collection where the order is not controlled by the programmer's interactions. Notably, a `Set` rejects duplicate items.

Wrappers and Autoboxing/unboxing

Wrappers and Primitives

Because reference types and primitive types are structurally different, there are situations where an otherwise generally applicable tool, such as a list, can't work directly with a primitive value. To address this, Java provides "wrapper" objects that represent a single, immutable, primitive value enclosed in an object.

Each of the 8 primitives has an associated wrapper. The names are mostly the same, except the first letter of the wrappers are capitalized. Also, the primitive `int` is wrapped by `Integer`, and the `char` is wrapped by `Character`.

In addition to containing a single, immutable, value, these classes also provide utility methods, for example facilitating conversions to and from textual representations in various bases, and comparisons.

```
double pi = Double.parseDouble("3.1415927");
```

Autoboxing and Auto-unboxing

Creating these wrappers, and extracting the value they contain can be done using the methods of the various wrappers, but the Java compiler can simplify and tidy the source by automatically creating the code to do this. This is called auto-boxing and auto-unboxing. This means that if we attempt to use a primitive where an object is needed, the compiler might help by creating the necessary conversion code for us.

```
Object obj1 = 99; // equivalent to Integer.valueOf(99);
Object obj2 = Integer.valueOf(99);
Integer obj3 = 99; // also equivalent to Integer.valueOf(99);

int x1 = obj3; // equivalent to obj3.intValue();
int x2 = obj3.intValue();

List<Integer> numbers = new ArrayList<>();
numbers.add(99); // adds Integer.valueOf(99)
```

Limitations of Autoboxing/unboxing

Not all conversions between primitives and references will be performed automatically.

When boxing, the basic type cannot be simultaneously converted. These fail:

```
Long obj3 = 99; // cannot convert int to long and box at the same time
Object obj1 = 99; int x1 = obj1; // Objects might not be numbers!
```

Also, while an `int` can be assigned to a `long` (the primitive `int` value definitely fits in the larger `long`), there is no assignment compatibility between `Integer` and `Long` object types. They are both, however, subtypes of a parent type called `Number`

```
Integer x = 99;
Long l = x; // fails
Number n = x; // this is OK
```

The Map Data Structure

The Map Data Structure

A map (sometimes called a dictionary, a table, or an associative array) has a set of unique keys, and paired with each key is a value (values can be duplicated). The structure is set up so that searching for any particular key is fast, as is pulling the matching value for a key. However, searching through the values directly, while possible, is not particularly fast. In this way, a map provides a lookup mechanism that's easy to use if a meaningful system of keys can be determined.

In Java's APIs `Map` is an interface that describes access to structures that allow adding and removing key-value pairs, and also changing the value associated with a key, in addition to reading the value associated with a key. In this sense, one can liken a map to a database table with a single, unique, primary key, and a single value in each row.

Two common `Map` implementations exist, these are `HashMap` and `TreeMap`. Elements to be added to a `HashMap` must provide valid `equals` and `hashCode` methods. Those to be added to a `TreeMap` must be provided with some kind of ordering mechanism.

Example Using a Map

A map might be used to store phone numbers (as Strings in this example) for quick lookup against a person's name. Remember that keys must be unique, so this will quickly fail where duplicate names exist.

```
Map<String, String> numbers = new HashMap<>();
numbers.put("Fred", "211 309 0929");
numbers.put("Jim", "639 518 2213");
numbers.put("Sheila", "627 153 8365");
String freds = numbers.get("Fred"); // freds = "211 309 0929"
numbers.put("Jim", "535 200 1094"); // Jim has a new number
String alexs = numbers.get("Alex"); // not found, alexs == null
```

Notice that looking up a non-existent key is not an error, but the request returns the sentinel value `null`. Be careful to test for this possibility before doing any processing on the retrieved value.

Iterating a Map

A Java `Map` does not implement either `Iterable` or `Collection`, but it's possible to get a `Set` of the keys, a `Collection` of the values (remember duplicate values are permitted, but not duplicate keys) or a `Set` of 'key-value pairs' represented using `Map.Entry` objects. Any of these can be iterated. So, using the numbers `Map` from earlier we can access all the key-value pairs and print the contents of our phone-number dataset like this:

```
Map<String, String> numbers = // as previously initialized
for (Map.Entry<String, String> pair : numbers.entrySet()) {
  System.out.println(pair.getKey() + " has number: " + pair.getValue());
}
```

Introduction to Java I/O

Byte and Character Sequences

Java distinguishes 8 bit byte data from textual data. Text data is represented in the JVM using a 16 bit Unicode format. Both bytes and characters are commonly treated as sequences of data items. In Java there are a number of classes that handle byte data and a number that handle character data. There are also conversion tools between the two.

Byte data is commonly handled using `InputStream` and `OutputStream` variations.

- Note that `InputStream` and `OutputStream` are *unrelated* to the much newer "Streams" API, which is a functional programming "monad-style" data processing framework.

Character (`char`) data is commonly handled using `Reader` and `Writer` variations.

`InputStream`, `OutputStream`, `Reader`, and `Writer` variations are designed to be able to plug into one another to create a pipeline of processing, for example reading byte data encoded in, say, ASCII, converting it to Unicode 16 bit data, and then buffering it.

Typical Configuration Reading Byte-oriented Text

To read textual data from a file, several steps exemplify the general case (although there are shortcuts available too, but these are less educational). Note that this code throws checked exceptions which must be addressed for the code to compile.

```
InputStream inS = System.in; // get the raw keyboard data byte stream
// then convert bytes to Unicode (a Reader)
InputStreamReader inR = new InputStreamReader(inS);
// add buffering and a convenient mechanism for reading a line at a time
BufferedReader bR = new BufferedReader(inR);
String line = bR.readLine(); // readLine is a feature of BufferedReader
System.out.println("Data read: " + line)
```

Note that by default the `InputStreamReader` will convert bytes using the platform's encoding, but this can be overridden by specifying an argument to the `InputStreamReader` constructor

Reading a Byte-oriented Text File

One benefit of the modular approach taken by Java's basic IO APIs is that it's easy to change the source of data. To convert the previous example to reading a file, we can simply change the InputStream.

```
// get the data byte stream from a file
InputStream inS = new FileInputStream("data.txt");
// then convert bytes to Unicode (a Reader)
InputStreamReader inR = new InputStreamReader(inS);
// add buffering and a convenient mechanism for reading a line at a time
BufferedReader bR = new BufferedReader(inR);
String line = bR.readLine();
System.out.println("Data read: " + line);
```

IO Utilities

Although the base facilities are very flexible, there are many utilities that perform multiple steps at once, resulting in cleaner code.

A file can be opened as a `Reader` directly:

```
FileReader fR = new FileReader("data.txt");
BufferedReader bR = new BufferedReader(fR);
System.out.println("Data read: " + bR.readLine());
```

And a `BufferedReader` can be obtained from a file directly:

```
BufferedReader br = Files.newBufferedReader(Path.of("data.txt"));
System.out.println(br.readLine());
```

A Simple Network Server

Java can treat low level network traffic in the same way as any other byte or character stream, this server accepts one connection, reads a single line of text, then echos it back with a prefix.

```
try ( ServerSocket ss = new ServerSocket(9000);
      Socket s = ss.accept(); ){
    BufferedReader iR =
        new BufferedReader(new InputStreamReader(s.getInputStream()));
    PrintWriter pw =
        new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
    String message = iR.readLine();
    System.out.println("Received message " + message);
    pw.println("I heard you say " + message);
    pw.flush();
  } // try-with-resources closes everything
```

A Simple Network Client

This network client opens a connection, sends a single line of text, then prints the response before closing the connection.

```
try (Socket s = new Socket("127.0.0.1", 9000)) {
  PrintWriter pw =
      new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
  BufferedReader iR =
      new BufferedReader(new InputStreamReader(s.getInputStream()));
  pw.println("It's a lovely day!");
  pw.flush();
  String message = iR.readLine();
  System.out.println("Reply is: " + message);
} // try-with-resources closes everything
```

Notice that the syntax and mechanisms of processing of the input and output is the same for the `Socket` obtained in the server as for the `Socket` obtained in the client.

Introduction to Classes and Objects

Basic Class Declarations

The general form of a Java class declaration is exemplified here:

```
public class Customer {
  private String name;
  private String id;
  private long amountOwing = 0;

  public Customer(String name, String id) {
    this.name = name;
    this.id = id;
  }

  public void addDebt(long amount) {
    amountOwing += amount;
  }
}
```

Classes do not have to be `public`

Fields are usually declared `private`, and can be initialized or not as preferred. These declarations do not have to be at the top of the file

A constructor does not declare a return type, takes its name from the class, and initializes a new object in the special variable `this`

Methods can be static or instance. Without the keyword `static`, a method is an instance method, and automatically receives a reference to the object to be worked on in the special variable `this`

Instantiating an Object From a Class Definition

Given a class definition such as the preceding one (whether part of some library, or coded by the programmer), we can build an object from the template by using the `new` keyword.

The use of `new` is followed by the name of the class to be instantiated, and then parentheses containing argument expressions that must match the type and number of the arguments to one of the class's constructors. So for the preceding example we could say:

```
Customer c = new Customer("Fred", "1Z9-302-0111");
```

A class is permitted to declare multiple constructors, provided the formal parameter list differs in type-sequence, so the `Customer` could have another constructor looking like this:

```
public Customer(String name, long credit)
```

OK, String, long differs from String, String

but not one like this:

~~`public Customer(String name, String address)`~~

NO, String, String conflicts!

Constructors and the Operation of `new`

The code referred to as a constructor looks very much like a method, but there are crucial differences:

- A constructor must not declare a return type. If it tries, it simply becomes a regular method with the same name as the class. It does *not* cause an error, which can be very confusing.
- The name of the constructor must be exactly that of the class
- A constructor receives an implicit parameter called `this`, which is the object that has been created by the `new` operation, and which should now be initialized.
    - Note that in this sense, the constructor in Java is not really a *constructor*, rather it's an *initializer*. The construction part–that is to say the *allocation of memory*–is entirely handled by the `new` keyword, and the programmer cannot influence this part of the process.
- A constructor can use the `return` statement to exit before the end of the body, but may not return any value.

Overloaded and Default Constructors

A class can contain code for multiple constructors provided the argument lists differ in terms of type-sequence.

If (and *only if*) the source-code contains *no constructors at all*, then a *default constructor* is created by the compiler. This takes zero arguments and has the equivalent of an empty constructor body.

One constructor can delegate to another constructor (for example, to avoid code duplication) using the word `this` as an invocation. Such delegation *must occur first* in the constructor body:

```
public class Date {
  private int day, month, year;

  public Date(int day, int month, int year) { … }

  public Date(int day, int month) {
    this(day, month, 2022); // delegate to above constructor
  }
…
```

The Initialization Process

Initialization of an object takes place in a predictable sequence:

- Memory is allocated and zeroed for the new object. If this cannot complete fully, an exception is thrown.
- Control is transferred to the constructor that matches the argument type sequence provided to the invocation of `new`.
- If the constructor starts with `this(...)` control delegates to the matching overloaded constructor, otherwise, control delegates to the constructor of the parent class
- If control returns from the superclass constructor, then instance initializers are run from top to bottom
- After return from the constructor delegated to, the body of this constructor executes.

Note that either `this(...)`, or superclass constructor delegation, will *always* be the first thing in the constructor. If nothing explicit is done, the compiler inserts delegation to the zero-argument parent class constructor.

Distinguishing `static` and Instance Elements

A class is represented in memory at runtime by an instance of an object of type `java.lang.Class`. This object provides a runtime "home" for any static fields, and also for the code associated with the class

Each static field declared in a class creates a single element of storage. Instance fields, by contrast, exist on a one element per object basis. So, if the `Date` class looks like this:

```
public class Date {
  private int day, month, year;
  public static int daysInJanuary = 31;
}
```

and we create four `Date` objects, then there will be four values each of `day`, `month`, and `year`, but only one of `daysInJanuary`.

Therefore, when referring to a `day`, we must qualify which object's `day` field we are referring to.

Qualified Identifiers

In the preceding discussion, a field called `day` can be selected with a prefix, so if we have two date objects representing January 1st 2022 and January 2nd 2022 respectively, declared like this:

```
Date first;
Date second;
```

then the reference `first.day` will have the value 1 and `second.day` will have the value 2.

To refer to the static field we can use these qualified forms:

```
packagename.Date.daysInJanuary
Date.daysInJanuary
```

The first form is usually unnecessarily verbose, and the second is sufficient if the code is in the same package, or if the `Date` class has been imported

The `this` Identifier

In any constructor or instance method, there is an implicit variable `this` which cannot be `null`. The `this` value refers to an object of the enclosing class type. It's entirely valid to use `this` as a prefix for identifying a particular object, for example indicating which object contains the field we want to select.

In the case of a constructor, `this` refers to the object that was allocated and zeroed by the action of the `new` keyword.

In the case of an instance method, `this` refers to the object that was the prefix of the method invocation. So, if we have the instance method `addDebt(int amount)` shown earlier, and we have a `Customer` object referred to by the variable `fred`, then the invocation **`fred.`**`addDebt(99)` results in the `this` value inside the method referring to the same object as referred to by `fred` on the outside.

Accessing Instance Elements in `static` Methods

It is common to hear an assertion on the lines of "you can't access instance fields from `static` methods". This is *imprecise and misleading*.

The actual restriction being mis-described is that a `static` method does not have a `this` value. However, a static method that has a reference to an object is no more or less capable of accessing a field of that object than an instance method would be.

As a side note, be aware that the access control mechanisms described in the section on encapsulation might prevent *either type* of method from accessing a field of a particular object.

Qualified and Unqualified Elements

In Java, any reference to a simple identifier, for example `theName`, will be searched for according to a general sequence (note that the search quits at the first success):

- If a local identifier (defined in the current method) exists by that name, the compiler determines that it is the target of the identifier.
- Otherwise, if the prefix `this` exists in the current scope, then the compiler will try qualifying the identifier with with it in the current class.
- The compiler will attempt to prefix with the classname. Note that it would be a compilation error to have two identifiers with the same name in the same class, so the order of checking the `this` prefix and checking the class name prefix cannot be relevant.
- If neither of these prefixes find an element, then the search proceeds up the class hierarchy, looking for either a static or instance element one level at a time.

Standard Methods of All Classes

In Java certain methods definitely exist in all classes, even if they're not explicitly coded in the source. These methods are inherited from parents, notably the universal parent class called `java.lang.Object` (inheritance is a later topic). Three methods in particular should be recognized:

- `public String toString()` – this method creates a textual representation of an object
- `public boolean equals(Object o)` – this method returns true if it decides that the argument object is equivalent to this object
- `public int hashCode()` – this method returns an integer which *must* be the same for any two objects that are considered equivalent by their equals methods, and *should* differ otherwise. It's used internally by the collections APIs for accelerating lookup in some of the data structures.

Note that although these methods will definitely exist, unless they are coded in the particular class of interest, their behavior might not be useful.

Overview of Record Types

Java 16 finalized the introduction of record types. Records provide a more succinct means of declaring classes that are intended to carry simple aggregated data, and which are expected to be immutable. Notable characteristics of records include:

- All fields, called "components" of a record must be declared using a special syntax, and their values will be `final`. This imposes a kind of immutability. Specifically, the value cannot ever change, but if the value is an object reference, this does not necessarily prevent the object's contents being changed.
- "Accessor methods" that have the same name as the fields are automatically created to provide for reading the data contained in the record
- `toString`, `equals`, and `hashCode` methods are provided by the compiler, and provide read-access to the fields.
- Record classes are implicitly `final` (they cannot become parent classes)
- Record classes cannot have explicit parent classes

Example of Record Types

Here's an example of a date represented using a record type:

```
public record Date(int day, int month, int year) { }
```

Given this, the following code would work:

```
Date today = new Date(22, 2, 2022);
System.out.println(today); // prints: Date[day=22, month=2, year=2022]
System.out.println("the year is " + today.year());
Date alsoToday = new Date(22, 2, 2022);
System.out.println("today.equals(alsoToday) "
  + today.equals(alsoToday)); // prints true
```

Encapsulation

Introducing Encapsulation

It's generally easier to use any device if it has a few means of interaction, and those interactions do one thing that's very evident, and does not have unexpected consequences.

It's also easier to use a device if interactions never "break" the device, but instead the device simply rejects attempts to do things that make no sense.

Further, a device of this kind usually has a metal box around it, to prevent users from messing with the internals, which would almost certainly damage the device in ways that would break the expectations created in the paragraphs above.

In software, we use the term "encapsulation" to describe the act of creating a logical "metal box" around our objects, thereby preventing uncontrolled interactions and rejecting attempts to perform nonsensical actions. Of course, we should also design the interactions (the application programming interface or API) so that it's very evident what any given interaction will do, and we should avoid unexpected side effects in those interactions.

Implementing Encapsulation

Implementing encapsulation requires both syntax and design. What operations are provided, and what they do and do not do, is a design question. Preventing access to the internal implementation is a syntax issue.

Java provides four levels of access control, `public` and `private` are the most important two:

- `public` means "accessible anywhere in this module" (and in specific situations in other modules)
- `private` means "accessible within the top-level curly brace pair surrounding this declaration"

For encapsulation to be robust, any *mutable* field in an object or class should be private. Any *immutable* element may reasonably be public provided it's always properly initialized.

Methods can have any accessibility. Those intended for general use are likely to be public. Those intended solely for internal support should likely be private.

Intermediate Accessibility

Java provides two intermediate levels of accessibility:

- Default accessibility (no accessibility modifier) means that an element is accessible anywhere within the same *package*. In well encapsulated objects, this can be useful for methods that are safe for external access, but really are only intended for use by closely cooperating classes.
- `protected` elements are accessible anywhere that a default access element is accessible, but are also accessible in a subclass providing the access is made using a reference expression that has the subclass type, rather than the parent type. (In a sense this means "subclasses can access this in objects of the subclass, but not in parent objects").

These intermediate access levels are typically used in creating libraries. It has been suggested that their use either implies a very good design, or a very bad one. As a general guide, using them for mutable data seems particularly dubious.

Implementing Encapsulation

A general guide is to make all fields private, and ensure that all constructors, and any methods that change data, validate the result will be in a valid state. For example, a `Date` class might have these elements:

```
public class Date {
  private int day, month, year;

  public Date(int day, int month, int year) {
    if (!makesValidDate(day, month, year)) throw new IllegalArgumentException("Bad values");
    this.day = day; this.month = month; this.year = year;
  }

  public void setDay(int day) {
    if (!makesValidDate(day, this.month, this.year)) throw new IllegalArgumentException("Bad
day");
    this.day = day;
  }
  // static method makesValidDate and more utilities, e.g. toString
}
```

Inheritance

Implementation Inheritance

Most object oriented languages provide syntactic support for the idea of implementation inheritance. This idea allows building a class that effectively starts from another class and then makes variations, perhaps adding extra features, or changing the implementation of others.

When such a mechanism is provided, it's usual to provide another feature that might be called "substitution". In this model, if we say that a class called Holiday is a Date with the extra feature of having a name, then we also say that a Holiday is a Date. The effect of this is that a Holiday object can be used as a substitute in any situation where a Date is expected.

Java supports all of this in a single-implementation-inheritance model. That restriction says that a class can have only a single parent (indeed, all classes except Object itself have exactly one parent).

Implementing Inheritance in Java

The inheritance syntax in Java uses the keyword extends. To show the simplest form here's a class definition for a Date and a Holiday. In this situation, a Holiday object is assignable to a Date reference, and has fields `name`, `day`, `month`, and `year` (although the accessibility of each of those is severely restricted, which we'll address shortly):

```
public class Date {
  private int day;
  private int month;
  private int year;
}

public class Holiday
    extends Date {
  private String name;
}
```

Given this method declaration:

```
public static void useDate(Date d) {}
```

then the following code is valid–note that the invocation of `useDate` assigns the `Holiday` reference `h` to the actual parameter of `useDate` which is of type `Date`.

```
Holiday h = new Holiday();
useDate(h);
```

Literally, a `Holiday` "is a" `Date` with some extras.

Access Control in Inheritance – 1

In the preceding example, all the fields are marked `private`. This means that methods in the `Holiday` class cannot access the fields `day`, `month`, or `year`. This might be troublesome. Two main approaches are possible. First is to make the parent class fields protected access :

```
public class Date { protected int day, month, year; }

public class Holiday extends Date {
  private String name = "unknown";

  public void show() {
    System.out.println("day: " + day + " month: " + month
        + " year: " + year + " name: " + name);
  }
}
```

Access Control in Inheritance – 2

In Java, protected access is *less* protected than default access, and default access allows access from anywhere in the same package. This might not be sufficiently strict control, in which situation, accessor methods might be employed to give controlled access to child classes and more. In this example, read-only access is publicly available. Because changes are prevented, encapsulation is secure:

```
public class Date {
  private int day, month, year;

  public int getDay() { return day; }

  public int getMonth() { return month; }

  public int getYear() { return year; }
}
```

Access Control in Inheritance – 3

If modification ability is required, mutator methods may be added. Such methods should protect the object by enforcing rules for the object's integrity and were described in the section on encapsulation. Here's the example in context:

```
public class Date {
  private int day, month, year;

  public int getDay() { return day; }

  public void setDay(int day) {
    if (!makesValidDate(day, this.month, this.year)) {
      throw new IllegalArgumentException("Bad day");
    }
    this.day = day;
  }
}
```

Method Overriding

When a subclass is created, it automatically has all the methods of it's parent. Those that are private are of course inaccessible from the child, but they're there anyway and can be available indirectly from the directly-accessible code of the parent. Static methods can be accessed too, although for best readability, this should generally be done using the parent class prefix. Static methods can be replaced with compatible methods in the child if desired, but the two remain distinct methods, and if the explicit class prefix is used, there's much less chance for surprises.

Instance methods that have compatible signatures exhibit a special behavior, known as overriding. In this situation, the child class's method definition will be used in preference to the parent class's definition, for any object of the child type. Importantly, this is the case even if the reference used is of the parent type.

Method overriding only occurs with non-private instance methods in parent classes for which the child class has a method definition that has the same name and formal parameter type-sequence as the original.

Method Overriding Example

In this example, the show method fo Date is overridden in the Holiday class.

```
public class Date {
  protected int day = 1, month = 3, year = 2022;
  public void show() {
    System.out.println("d: " + day + " m: " + month + " y: " + year);
  }
}

class Holiday extends Date {
  public void show() {
    System.out.println("Holiday on day: " + day + " month: " + month);
  }
}

Date d = new Holiday();
d.show();
```

In this example, the invocation `d.show()` prints "`Holiday on…`" because the `Date` reference `d` refers to a `Holiday` instance.

Ensuring Overriding

If a child method differs, even slightly, in its name or argument type sequence it will *not* be an override, but will be a new method, or perhaps an overload. Importantly, no error message will warn us of the mistake we made. To address this, if a method is intended to be an override it can be annotated with `@Override`. If the method is indeed an override, this has no effect, but if it's not an override, an error is issued:

```
public class Parent {
  public void doStuff(int x) {}
}

class Child extends Parent {
  @Override public void doStuff(int a) {} // valid-the name of the parameter is irrelevant
  @Override public void dostuff(int x) {} // invalid-different spelling (capitalization)
  @Override public void doStuff(Integer x) {} // invalid-different parameter type
  @Override public void doStuff(int x, int y) {} // invalid-"int" differs from "int, int"
}
```

Rules for Overriding Methods

Overriding must not break the "substitutability" of the child class. Recall that an instance of a subclass "is an" instance of the parent class. This means that replacement methods (overriding methods) must not cause nasty surprises when substituted. Three key rules apply:

- Overriding methods must not be less accessible. If they could be, they might not be accessible in places where they're expect to be so.
- Overriding methods must not declare they throw checked exceptions that are not declared to be possible from the parent method. If they could, they might bypass the checking system.
- Overriding methods must return a type that's "compatible" with the type returned by the parent.
    - For primitive types, this must be the *exact same type*
    - For reference types, the return type must be assignment compatible with the parent method's return type.

Generalized Assignments and Casting – 1

Because an instance of a subclass is considered also to be an instance of the parent class, it's possible to have a reference of the parent type that actually refers to an object of the child type. (We saw this previously). If we suspect this might be the case in a piece of code, we can test the situation using the instanceof operator:

```
Date d = new Holiday();
System.out.println(d instanceof Holiday); // prints true
```

Using the `Date` type reference `d`, we can't access any features that are unique to a `Holiday` object. But, if we know that a reference expression refers to an object of the child type, we can create a new expression that has the child type by using a *cast*. Such a cast should generally be protected with an instanceof test:

```
if (d instanceof Holiday) {
  Holiday h = (Holiday)d; // (Holiday) is the cast
  // use h to access Holiday-specific features
}
```

Generalized Assignments and Casting – 2

The two part form just shown, involving a test and then a cast, is somewhat cumbersome and error prone. It's generally too easy to forget the test. Because of this, Java 16 introduced a new syntax called "pattern matching" that simplifies the test and cast to a single syntactic element:

```
// this form assigns d to h only if d passes the instanceof test
if (d instanceof Holiday h) {
  // use h to access Holiday-specific features
}
```

The scope rules for h are "smart", that is, the compiler will make `h` available only in the places that it can prove `h` will definitely have been assigned.

```
if (!(d instanceof Holiday h)) { /* h is not accessible here */ }
else { /* use h here */ }
```

Using `final` with Variables

A variable (field or method local) can be marked final. If this is done, the compiler will ensure that it is assigned to exactly once. If it's a field, this must be done prior to completion of all possible paths through the constructors. Whether the variable is a field or a method local it becomes the nearest thing that Java offers to a constant, but it's important to realize that this is not reliably the same as a constant.

- If the final variable has primitive type, then it really is a constant value
- If the final variable refers to an immutable object type (such as a String) then again, it is a constant value
- However, if the final variable refers to a mutable object type (such as a StringBuilder) then it can never be changed to refer to another object, but the object itself may have its contents changed.

Using `final` with Methods and Classes

The modifier `final` can be applied to a method, in which case it prohibits overriding that method.

The modifier `final` can be applied to a class, in which case it prohibits any class from subclassing that class.

- Subclassing sometimes has unexpected consequences, so if you haven't specifically thought about the effect that a subclass might have, particularly on the integrity and encapsulation of your parent type, it's not a bad idea to mark your class final. If the need to subclass it arises later it's trivial to remove this word after you've given the necessary thought to the proposal.

A method local variable that is provably assigned exactly once is considered to be *effectively final*. Some other syntax elements require that a variable should be final or effectively final, and in this case, it's sufficient that exactly one initializing assignment is made for the requirement to be met.

Inner and Nested Classes

Nested Types

With a few restrictions, Java allows classes, and in fact types in general to be declared inside other types. These might be referred to as *nested*, or *inner*, or sometimes even *member* types.

One reason for doing this derives from the meaning of the keyword `private`. Recall that this means "accessible anywhere inside the *enclosing top-level curly braces*". Consequently, a nested type has full access to the private members of the enclosing type, and also that the enclosing type has full access to the private members of the nested type. Sibling nested types also have full access to each other. Nested typing is therefore a powerful way of granting privilege to a specific class or small group of classes, and is more robust than using packages for this effect.

If privileged access is all that is needed, a `static` nested type, or types, should be created. These classes will share privileged access, but nothing more. The term "nested" is generally reserved for static types inside other types.

Inner Classes

Sometimes a design calls for a "has a" relationship such that one class has a reference to, and interacts with, an instance of another class. If this is a one-to-one, essentially unidirectional, relationship and privileged access is required, then an instance inner class might be the best approach.

An instance inner class benefits not only from the privileged access inherent in any member class relationship, but also has implicit access to a particular instance of the enclosing class.

To build that "has a" relationship, the inner object must be constructed in the context of an instance of the outer class. This can be done implicitly or explicitly.

Declaring and Instantiating Instances of Inner Classes

An inner and outer class pair might look like this:

```
class Outer {
  private String name = "Outer"; // instance field!
  class Inner {
    private String myName = "Inner";
    @Override public String toString() {
      return "I'm " + myName + " inside " + name; // name refers to a field in the Outer object
    }
  }
}
```

Given the above class declarations, these instantiations will succeed

```
Outer o = new Outer();
Inner i = o.new Inner(); // succeeds, object i has a reference to object o

Inner i1 = new Outer().new Inner(); // succeeds
```

But this one would fail (*unless* it's in inside instance method of the Outer class)

```
Inner i2 = new Inner(); // fails, no Outer context
```

Lambda Expressions

Introducing Lambda Expressions

In an object oriented language, an object is a container of state (data) and behavior (methods/functions). This means that if we pass an object as a method argument, we pass not simply data, but also code that could be executed. Some (very powerful) design patterns make use of this. Those that do often need no more than to simply pass a single method. In Java, we can define an interface that describes a single method, and then implement that interface, and pass an object of that implementation. This has been possible since the very earliest versions of Java.

More recently, the power of this approach has pushed Java to a specialized syntax that expresses this goal with much more succinct code. This syntax is the "lambda expression".

Java's lambda expression is strongly and statically typed, and is required to become an object that implements an interface that's unambiguously known at compilation time, and which defines exactly one abstract method. The lambda expression is a brief description of the elements necessary to provide an implementation of that one method. The compiler takes this skeleton information and builds a class behind the scenes, and instantiates it.

A Behavioral Object *Without* Using a Lambda Expression

A common use of these "behavioral objects" is the implementation of a `Comparator`, used in sorting. Here is a class that implements this interface for sorting Strings, and a use of it:

```
class StringLengthComparator implements Comparator<String> {
  @Override
  public int compare(String o1, String o2) {
    return Integer.compare(o1.length(), o2.length());
  }
}
```

It can be used like this:

```
List<String> ls = new ArrayList<>(List.of("Alice", "Bob", "Fred"));
ls.sort(new StringLengthComparator());
System.out.println(ls);
```

Using a Lambda Expression for the Same Effect

Implementing the exact same behavior as the previous example, but using a lambda expression, looks like this:

```
List<String> ls = new ArrayList<>(List.of("Alice", "Bob", "Fred"));
ls.sort(
  (String o1, String o2) -> {
    return Integer.compare(o1.length(), o2.length());
  }
);
System.out.println(ls);
```

Notice that the core of the method from the previous example has been used, but the supporting "syntactic scaffolding" that defined the `StringLengthComparator` class and then instantiated it has been removed. Also note the use of the arrow symbol between the method's formal parameter list and the method body.

Further Simplifications for Lambda Expressions – 1

The lambda expression can be further simplified in this case. Taking this starting point:

```
ls.sort((String o1, String o2) -> {
  return Integer.compare(o1.length(), o2.length());
});
```

Notice that the context of the ls.sort method demands a Comparator<String>--this is crucial to the lambda syntax, and is how the compiler decides what kind of class to build and instantiate.

The compiler also knows that in this case the formal parameter list must take two Strings. This specification can be omitted, producing:

```
ls.sort((o1, o2) -> {
  return Integer.compare(o1.length(), o2.length());
});
```

Note this formal parameters must be either all type-specified, all devoid of type specifications, or entirely declared using the `var` pseudo-type.

Further Simplifications for Lambda Expressions – 2

If, as in this case, a lambda body simply returns the evaluation of an expression, we can leave out the method body's curly braces, and the return keyword and its associated closing semi-colon. Now the body is replaced with a simple expression:

```
ls.sort( (o1, o2) -> Integer.compare(o1.length(), o2.length()); );
```

If a lambda implements an interface that declares a single abstract method that takes exactly one formal parameter, and we choose to leave off the type of that parameter entirely, we can also leave off the parentheses around the formal parameter list:

```
Predicate<String> ps = s -> s.length() > 3;
```

Rules for Lambda Expressions

Lambda expressions have some rules that must be adhered to:

- They can only be used where the compiler can determine that an expression is required
- The type of that expression must unambiguously be of an interface type
- The interface must declare exactly one abstract method
- The argument list of the lambda expression must be compatible with the argument list of the abstract method
- The return type of the lambda expression must be compatible with the return type of the abstract method

Introduction to the Module System

Some Key Goals of the Module System

- Package-level access control was easily subverted because a class can simply be placed in that package to gain the access.
  - The module system prohibits a package existing in more than a single module.
- Public accessibility allowed access to the element from anywhere in the running JVM
  - The module system changes public access to mean anywhere the same module, unless the containing package is expressly exported from the module, and another module expressly requires the exporting module.
- By default reflection was permitted to break the protections even of private elements.
  - The module system prevents "cross-module" reflection unless expressly enabled.
- Dependencies between libraries were entirely ad hoc, and nothing in the source code documented these, nor could such dependencies be automatically extracted. Further, circular dependencies were not prevented
  - The module system makes dependencies explicit, and prohibits circular dependency.
- The module system also expands on the tools provided by Java for the provision and use of services

Exporting Packages

When a module intends to provide access to other modules the "entry points" to the module's facilities should be placed in a specific package (or packages). Those packages should then be exported from the module. This is done using a module-info.java source file which describes the module:

```
module my.providing.module {
  exports my.utility.package;
}
```

The `exports` directive makes it possible for other modules to use public features of the package named in the directive.

An exports directive can list specific modules that are permitted to access these features, preventing arbitrary modules from using them. This is done with the `exports` / `to` form:

```
exports my.utility.package to another.module, and.another.module;
```

Multiple `exports` directives can be added; only one package can be listed for each.

Requiring Modules

When a module wants to use the features of another module it must declare the fact in its module-info.java file:

```
module my.using.module {
  requires my.providing.module;
}
```

In this situation, all the packages that are exported from `my.providing.module` will be available in this module (unless they are exported using `exports to` without mention of this module).

The Module Graph

Because the module system requires that every module declare explicitly all its dependencies, it's possible for the JVM to build a directed acyclic graph of all the modules needed to run any given application at the moment of startup.

The graph is used to ensure the minimum necessary set of modules are searched when performing classloading. This has the effect of speeding up classloading, and hence application startup. If an application is packaged for distribution, it also allows a smaller binary to be built that only includes the modules that are actually needed.

Another, less desirable, consequence of this is that if legacy, non-modular, code is mixed with modular code–which often happens during migration of a large system to run under the module system–it's often necessary to "patch up" the module graph manually using command-line options for the purpose to ensure that all necessary code is available at runtime.

Another consequence of the prohibition on cyclic dependencies in the module system is that it becomes hard to migrate some code that has such cycles.

Controlling Reflection

By default the module system prevents code in one module from performing reflective access on another module. However, this can be enabled explicitly using the `open` or `opens` directives.

A module declared as open permits reflection on its contents from other modules:

```
open module my.reflectable.module { }
```

More commonly, reflection might be permitted on individual packages, in a manner somewhat parallel to the exporting of individual packages:

```
module my.reflectable.module {
  opens my.reflectable.package.one;
  opens my.reflectable.package.two to reflecter.moda, reflecter.modb;
}
```