Java Guide

By Richard Yin

Java code works very differently from Python; I will try to hide my frustrations in this guide. I'll assume you use PyCharm and are already well-versed in Python, and you are using IntelliJ as your Java IDE.

I frequently compare Java functions to Python functions. Sometimes, these comparisons are not 100% exact – they may handle fringe cases differently or are more strict/lax in their arguments, etc.

Also, my code might not be 100% correct, so…yeah.

Starting Projects

In Python, life is good.

You create a .py file.
You type code.
You run it in PyCharm.
You can see your variables saved in Pycharm's bottom-right tab.
You can open Python.exe or type in the Python console to test bits of code.

In Java, life's a lot harder.

You create a project folder, make an inner folder called src, and then create a .java file in that.
All .java files contain exactly 1 class - projects will have many, many .java files.
All Java code must be inside a class.
Class names must be the same name as its .java file.
When you run a file in IntelliJ, Java compiles every single .java file in your project first (slow), returns a single result, and exits.
There is no Pycharm console area for bugfixing after you run the file; you can only print results.

Python is an **interpreted language** – a special interpreter reads and executes your code line-by-line.
Java is a **compiled language** – all your code is converted to machine code first, which a processor executes.

The process of converting to machine code, or **building**, must be done every time before running.

- Compiling usually makes the code faster/more efficient than Python when running
- Type restriction bugs in methods/variables/element-based loops are checked while compiling
- If there's an error anywhere in any file (even if you never use that file), the project won't run.
- There's no line-by-line testing, so bugs are harder to bug-test than Python.
- Classes in a Java project can be called in other classes without importing/writing `file.className`

The reason everything's in a class is to encourage **class-based object-oriented programming**.
The reason Java's so strict is to enforce clarity and basic hygiene for *large, corporate software dev teams*.

Java is descended from the same family of languages as C, so languages like C# and C++ have very similar verbose syntax. So get used to it.

Visibility

Java **packages** are like folders in a file directory. They can be user-created or built-in, like this package:

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

Java cares a lot about visibility; here're four **accessibility modifiers** for methods/variables:

1) `private`: accessible only in the class where it was declared
2) `default`: accessible in the class and package where it was declared (ie. package-protected)
3) `protected`: same as default, but also accessible through child classes
4) `public`: accessible inside/outside the declared class (ie. anywhere)

Left unspecified, methods and variables are `default`.

Typing

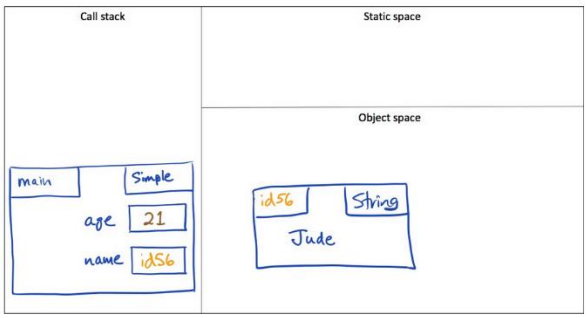
Python's a **weakly/loosely-typed language** – it auto-detects variable types, and you can change it.
Java's a **strongly/strictly-typed language** – every variable's type is explicitly said and unchangeable.

```
int num = 5;
String str = "Hi";
num = str;              // Error! Can't change types.
```

Python's a **dynamically-typed language** – variables are assigned types based on their value at the time
Java's a **statically-typed language** – all variable types are figured out before the code is even executed

Memory Model

Call Stack: Left, all currently-running methods
Object Space: Bottom-right, all objects
Static Space: Top-right, all static objects *(you'll see later)*



Type of Equality	Python	Java
Value	`==`	`.equals()`
Identity (id)	`is`	`==`

The Java value equality is a class method. The default `Object.equals()` is identity equality, but it's been overridden in most data types we work with to mean value equality.

By default, Java's `.clone()` method returns a shallow copy of an object.

Shallow Copy: When `var1` and `var2`'s ids are different, but their items' ids match.
Deep Copy: When `var1` and `var2`'s ids are different and their items' ids are different too.

Data Types

Primitive: A data type with no id or methods, referring to itself in the memory model.

- `byte` stores integers from –128 to 127
- `short` stores integers from –32,768 to 32,767
- `int` stores integers from –2,147,483,648 to 2,147,483,647
- `long` stores integers from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- `float` stores fractions with 6-7 decimal point accuracy
- `double` stores fractions with 15-16 decimal point accuracy
- `char` stores single characters/letters (eg. `'a'`, single-quotes only)
- `boolean` stores `true`, `false`. Type coercion doesn't happen in Java for booleans.

Reference: A data type that refers to an id. What you're used to from Python.

- `Object` same as Python object. Everything's a subclass of it.
- `String` any character/letter combination (eg. `"a"`, `"abc"`, double-quotes only)
- `Integer` object form of `int`
- `Double` object form of `double`
- `Character` object form of `char`

BTW, this isn't exhaustive. Primitive types are all-lowercase. Reference types start with upper-case.

```
int a = 5;
int b = a;
short c = 5;
a == c;     // true -> different types, but same memory address of 5
b == c;     // true
// a == c; and b == c; actually return nothing in Java. I just format it like this for clarity.
```

Reference types and classes can be instantiated with this verbose notation:

```
CoolClass hi = new CoolClass("Argument1", 2, false);
String str = new String("Hello");
Integer i = new Integer(5);
```

However, for reference types, there're shortcuts:

```
String str = "Hello";
Integer i = 5;
```

Autoboxing: Java behaviour that automatically converts primitive types to reference types and vice versa if the opposite type is given as a parameter.

```
int x = 2;
int y = 2;
Integer a = x;                // Autoboxing.
Integer b = new Integer(2);   // "new" allocates memory for a new object
Integer c = new Integer(2);   // Integer/Long/Character are "wrapper classes" of int/long/char
int d = new Integer(2);       // This is technically allowed, but why would you do this?

x == y;       // true  -> primitives store their value as their id
x.equals(y);  // error -> x is a primitive, has no methods!

a == x;       // true -> a is auto-unboxed to a primitive
a.equals(x);  // true

a == b;       // false -> because b has "new", which creates a unique id
a.equals(b);  // true

b == c;       // false -> because b and c have "new", which creates a unique id
b.equals(c);  // true
```

The reference types that have primitive type equivalents – like `Integer`, `Double` - are called **wrapper classes**. They exist because they contain useful methods, while the primitives can't have methods.

Strings are special cases that deserve mention for how unintuitive Java makes them.

Java	Python
`s1.charAt(2)`	`s1[2]`
`s1.substring(2, 4)`	`s1[2:4]`
`s1.length()`	`len(s1)`
`s1.replace('a', 'b')`	`s1.replace('a', 'b')`
`s1.replaceAll("abc", "def")`	`s1.replace('abc', 'def')`
`s1.split("-")`	`s1.split("-")`
`s1.concat(s2), s1 + s2`	`s1 + s2`
`s1.contains(s2)`	`s2 in s1`
`s1.indexOf(s2)`	`s1.index(s2), s1.find(s2)`
`var.toString(), String.valueOf(var)`	`str(var)`

You can find more Java String methods here. Note that `chars` and `Strings` do NOT autobox each other.

```
String s1 = new String("Hello");
String s2 = new String("Hello");
String s3 = "Hello";              // String has no primitive equivalent; this isn't autoboxing!
String s4 = "Hello";              // These strings are instead created in a "string pool".

s1 == s2;       // false
s1.equals(s2);  // true -> warning, if any of the strings is null, you get NullPointerException

s3 == s1;       // false -> similar behaviour as if autoboxing was happening!
s3.equals(s1);  // true

s3.equals(s4);  // true
s3 == s4;       // true -> it's as if s3/s4 are primitives, and s1/s2 are references!
```

String Pool: A special area of memory that Java *specifically assigns to strings* introduced without `new`. The pool has no ids, so value equality is identity equality there. `.intern()` sends strings to the string pool.

All primitives are immutable, so aliasing doesn't work.

- If you set `immut1 = immut2`, either can be set to different values without mutating the other.

Some reference types are also immutable, like `String`, but some are mutable, like `Stringbuilder`.

```
Stringbuilder s1 = new StringBuilder("Hi");
s1.append("wow");
s1.insert(0, "234");
s1.setCharAt(0, '5');
s1.reverse();

s1.toString(); // "wowiH435"
```

Python doctrine is to build a generalized catch-all version of data types and objects (But, specialized versions are available if you import different libraries)

Java doctrine is to build many versions of the same thing, each with specialized purposes (You've seen this with byte/short/int/long, and here it is with String/Stringbuilder)

Finally, here's some weird Java string behaviour that I don't understand.

```
// You'd expect .append() to return nothing, but it doesn't.
return new StringBuilder("Professor").append(" Horton"));
// "Professor Horton"

// Turns out you can instantiate an object right as it's being returned.
return new String("Hi");
// "Hi"
```

Variables and Methods

Here is the "official" formatting for variable assignment. The optional parameters aren't ordered.

```
(visibility) (final) (static) type varName (= …);
```

Visibility	Can be `private`, `default`, `protected`, or `public`. Left unspecified, it is `default`
Final	If `final` is written, the variable will be uneditable (a constant). The Python equivalent is to name the variable in all upper-case, but Python doesn't enforce it like Java.
Static	For class variables, if `static` is written, the variable initializes at program start. It's shared across all class instances, accessible even if the class isn't instantiated. *See the Classes section.*
Type	The variable's data type, which must be explicitly put in.
Name	Use `camelCase` in Java, not `pothole_case`
= …	Java will raise an error if you try to use an un-initialized variable. There are default values ONLY if you're creating a list of numbers/strings (see *Collections and Iterables* section): • `int: 0` • `boolean: false` • `String: null` (All classes are `null` if left unspecified.)

In Python, function and methods are the same thing. Method just means class function.
In Java, since all code is in classes, there are no functions, only **methods**! I might still say function, though.

```
(visibility) (abstract) (static) returnType funcName(arg1Type arg1, …) {
}
```

Visibility	Can be `private`, `default`, `protected`, or `public`. Left unspecified, it is `default`
Abstract	If `abstract` is written, then instead of the brackets {}, you just add ; at the end to signify an abstract method. In Python, the equivalent is raising a `NotImplementedError`.
Static	If `static` is written, the method initializes at program start. It's callable even if an instance of its class is not created.
Return Type	Java's equivalent of Python's return type `None` is `void`.
Name	Use `camelCase` in Java, not `pothole_case`. Usually should be a verb, but not always.
Arguments	The arguments are entered like a normal variable, with just the type and the name.

I believe everything static-related is stored in the Java memory model's **static space**.
The Python kind-of equivalent to `static` is just a variable/function that isn't in a class.

Static Variable/Method: A class variable/method with the descriptor `static`.
Instance Variable/Method: A class variable/method without the descriptor `static`.

Instance methods can call static variables and instance variables.
Static methods can call static variables but not instance variables.

```
// Suppose this code is inside a class
public String amog = "us";

/** Here's a Javadoc (Python equivalent of docstring). They belong outside a method/class:
 * @param sus the string to add to the end of amog.
 * @return the string amog but with sus added to the end.
 */
public static String makeSus(String sus) {
    return amog + sus;  // Will have an error.
}
```

Since you instantiate no class when you call a static method, no instance variable exists to be changed.

IntelliJ italicizes static variables/methods. Some common static methods you might see are:

```
double x = Math.cos(40);   /* Check out abs(), ceil(), exp(), log() (ie. ln), hypot() (finds
hypotenuse, given two sides), max(), round(), random(), rint(), toDegrees(), toRadians() */
String y = String.valueOf(2);

// Java's print method. It's so cumbersome, IntelliJ has a shortcut for auto-writing it: "sout"
// println() creates a line break. print() doesn't. System.err.…() can print error messages.
System.out.println("BTW you can't multiply ints by Strings in Java.");
```

Classes

```
(visibility) (abstract) (final) (static) class ClassName {
}
```

Visibility	Almost always (and by default), your class is `public`. Nested classes can be `private`
Abstract	If `abstract` is written, then Java knows it's an abstract class. *See the Inheritance section.*
Final	If `final` is written, the class cannot have child classes. `abstract` classes can't be `final`.
Static	Only nested classes can be `static`. There's a section on these later.
Class	Just write `class`.
Name	First letter should be capital, with new words starting with capitals. Should be a noun.

In Python, instance variables of a class always have `self.` at the front.
In Java, instance variables have `this.` at the front, but only if a similarly-named local variable is also used.

```
public class Country {
    private int population;              // Instance variable
    private static int totalCountries;   // Static variable, shared across all Country classes

    public int setPopulation(int population) {
        this.population = population;
        // Use this.… when there're a conflict of names. Otherwise, it's redundant.
    }
}
```

Constructor: The Java equivalent of Python's `__init__()`. This method have the same name as the class and have no return-type specified. They're usually public (except for classes with only static vars/funcs)

```
public class Country {
    private int population = 10000;      // Instance variable.
    // The constructor will auto-set population to 10000 on instantiation. Convenient!
    private static int totalCountries;   // Static variable, shared across all Country classes

    public Country(int population) {     // The constructor.
        totalCountries += 1;             // Total countries increase with each instantiation
        // It's good practice to write Country.totalCountries for static variables.
    }
}
```

Public Static Void Main (psvm): The Java equivalent of Python's `if __name__ == "__main__"`:, this code automatically runs when the project is executed. You usually put this in a separate main class of its own.

```
public class Start {
    public static void main(String[] args) {  // The method must be called exactly like this.
        // This is the easiest way to bugtest or try out small lines of code in Java.
        // IntelliJ autofills this for you if you write "psvm".
    }
}
```

The method must be `public` *(accessible from anywhere)*, `static` *(runnable without creating an instance of Start)*, `void` *(returning nothing)*, and called `main`. It takes an array of Strings, `args`, as its argument *(see the next section on Collections and Iterables)*, but IDK how that makes it run automatically haha.

Consider the code below. It seems redundant – why not just make the variables public?

```java
class School {
    private String name;

    public School(String name) {
        this.name = name;   // Constructor
    }

    public String getName() {
        return name;
    }
}
```

```java
class Student {
    private School school;

    public Student(School school) {
        this.school = school;   // Constructor
    }

    public School getSchool() {
        return school;
    }

    public void setSchool(School school) {
        this.school = school;
    }
}
```

Encapsulation: A feature of object-oriented design which includes bundling data with methods and hiding internal representation. In English, this means accessing variables indirectly with methods and making as many things private as possible. Why?

- Teammates building off your work won't see inner gunk.
- No one will mistakenly mutate the inner variables and mess things up.
- It communicates your intentions.
    - If you want a variable to be left alone, make it private and leave it like that.
    - But if you want it to be retrievable, create a public `getVar()` function: a "**getter**"
    - But if you want it to be mutable, create a public `setVar()` function: a "**setter**"
    - If you want to indicate a helper method, make it private.

The consequence of this school of thought is **boilerplate code** – ugly, repetitive code. Thankfully, IntelliJ provides tools to auto-generate the all-too-common "getters" and "setters".

Also, be aware of how aliasing works with class variables.

```java
School school1 = new School("UofT");
Student student1 = new Student(school1);

School school2 = new School(student1.getSchool().getName());
Student student2 = new Student(school2);

student1.getSchool().getName() == student2.getSchool().getName();       // true
student1.getSchool().getName().equals(student2.getSchool().getName());  // true
```

`student1` and `student2` are deep copies despite school names having identity equality: strings are immutable.

`==` is `false` for different instances of a class. `.equals()` is `false` unless `.equals()` is implemented for the class.

Overloading

Java doesn't support default argument inputs or typing annotations like `Union` and `Optional`. Compare:

```java
public int func(int cool) {
    return cool;
}
public void func() {
    System.out.println("Lemons");
}
```

```python
def func(cool: Optional[int] = None) -> Optional[int]:
    if cool is None:
        print("Lemons")
    else:
        return cool
```

Overloading: When you define the same method multiple times, each time with different inputs, so that it can handle different input types.

Collections and Iterables

Python provides one flexible implementation of the list, set, and mapping data types.
Java provides multiple, special-purpose implementations of the list, set, and mapping data types.

Object	Implements	Description
`Array`	`N/A`	An un-resizable list. Multiple dimensions. Runs fast.
`ArrayList`	`List`	Resizable.
`LinkedList`	`List, Queue`	Doubly-linked list. Resizable. Also works as a queue. Memory-heavy.
`PriorityQueue`	`Queue`	Elements are accessed in FIFO (first-in-first-out) order. Can prioritize
`Stack`	`Stack`	Elements are accessed in LIFO (last-in-first-out) order. Can push/pop.
`HashSet`	`Set`	Implemented with hash table. Add, remove, contain are $\Theta(1)$. Elements are unsorted. More memory-heavy than TreeSet.
`LinkedHashSet`	`Set`	Sorted HashSet based on LinkedList, with predictable iteration order. Tracks insertion order. Add, remove, contain are $\Theta(1)$.
`TreeSet`	`Set`	Implemented with trees. Add, remove, contain are $\Theta(\log n)$. Sorted, iteration order is predictable. Contains more powerful methods.
`HashMap`	`Map`	Implemented with hash table. No guarantee on iteration orders. $\Theta(1)$.
`LinkedHashMap`	`Map`	Sorted HashMap based on LinkedList, with predictable iteration order. Tracks insertion order. $\Theta(1)$.
`TreeMap`	`Map`	Implemented with trees. Sorted, iteration order is predictable. $\Theta(\log n)$

This isn't exhaustive, of course. Our main focus is Arrays, which are instantiated with weird syntax.

```
int[] nums1 = new int[5];
// Creates an integer array of length 5 consisting of the default values (ie. 0)
// Default size is 10 if not specified, for all list implementations
// Terrible notation warning – int[5] uses the same notation as list indexing, var[i]

char[] chars = {'a', 'b', 'c', 'd', 'e'};
// Creates a char array of length 5 consisting of 'a', 'b', 'c', 'd', 'e'
// A "shortcut". Terrible notation warning – why use {} when that should be for sets?

int[][] num2 = {{1, 2, 3}, {4, 5, 6}};
// Creates a 2-dimensional integer array with the specified numbers.
// You can think of it as a 2 x 3 matrix.
// Being "multidimensional" sounds fancy, but it's really just a list of lists.

nums[0];          // Equal to 0
nums1[0] = 500;   // Mutates index 0 to 500. Doesn't support slices or negative values.
```

Arrays are like Python lists, but much more restrictive:

- They may only contain one data type – the one they're defined on. This is true of all Java lists.
    - There're painful and expensive ways to get around this. *See the Casting section*
- They are not resizable. If you want resizability, use ArrayLists.

If you print an array, you'll get something like "[Ljava.lang.String;@6a5fc7f7" – which isn't right. You have to print `Arrays.`*`toString`*`(array)` instead.

There's a lot more annoying type-enforcement behaviour with arrays when it comes to iterating and indexing through Java lists. See *Generics, Polymorphism, Casting sections*

Next page has a non-exhaustive collection of methods for the different collections. First up is lists.

Java	Python
`lst.add(item)`	`lst.append(item)`
`lst.add(3, item)`	`lst.insert(3, item)`
`lst.addAll(collection)`	`lst.extend(collection)`
`lst.clear()`	`lst.clear()`
`lst.clone()`	`lst.copy()`
`lst.contains(item)`	`item in lst`
`lst.containsAll(collection)`	`all(x in lst for x in collection)`
`lst.equals(other)`	`lst == other`
`lst.isEmpty()`	`len(lst) == 0, lst == []`
`lst.get(4)`	`lst[4]`
`lst.remove(4)`	`lst.remove(4), del lst[4]`
`lst.remove(lst.size() - 1)`	`lst.pop()`
`lst.remove(item)`	`lst.pop(item)`
`lst.removeAll(collection)`	`[x for x in lst if x not in collection]`
`lst.retainAll(collection)`	`[x for x in lst if x in collection]`
`lst.set(4, item)`	`lst[4] = item`
`lst.size(), lst.length`	`len(lst)`
`lst.subList(2, 4)`	`lst[2:4]`
`List.of(2, 3, 4)`	`[2, 3, 4]`

Here're the same for sets. Non-hashing Java sets can contain lists and dictionaries, unlike Python sets.

Java	Python
`s.add(item)`	`s.add(item)`
`s.addAll(collection)`	`s.update(collection), s.union(collection)`
`s.clear()`	`s.clear()`
`s.clone()`	`s.copy()`
`s.contains(item)`	`item in s`
`s.containsAll(collection)`	`all(x in s for x in collection)`
`s.equals(other)`	`s == other`
`s.isEmpty()`	`len(s) == 0, s == set()`
`s.remove(item)`	`s.remove(item), s.discard(item)`
`s.removeAll(collection)`	`s.difference(collection)`
`s.retainAll(collection)`	`s.intersection(collection)`
`s.size()`	`len(s)`
`s.toArray()`	`list(s)`
`Set.of(2, 3, 4)`	`{2, 3, 4}`

Here's the same for maps. Non-hashing maps can contain lists and map keys, unlike Python dictionaries.

Java	Python
`m.clear()`	`m.clear()`
`m.clone()`	`m.copy()`
`m.containsKey(item)`	`item in m.keys(), item in m`
`m.containsValue(item)`	`item in m.values()`
`m.equals(other)`	`m == other`
`m.get(key)`	`m[key]`
`m.isEmpty()`	`len(m) == 0, m == {}`
`m.keyset()`	`m.keys()`
`m.put(key, value)`	`m[key] = value`
`m.putAll(other)`	`m.update(other), {**m, **other}, m | other`
`m.remove(key)`	`m.pop(key), del m[key]`
`m.replace(key, value2)`	`m[key] = value2`
`m.size()`	`len(m)`
`m.values()`	`m.values()`
`Map.of(1, 2, 3, 4)`	`{1: 2, 3: 4}`

Some of these method equivalences are not exact (slightly different input/return types, some have more restricted uses), so you should probably look through the Java documentation.

Only Arrays are allowed by default. If you need any more data types, you have to import from `java.util`.

```
import java.util.ArrayList;  // Import ArrayList from java.util. IntelliJ can auto-suggest it
import java.util.*;          // Import everything from the default package java.util.
```

Control Structure

The keyword `break` exits the innermost loop it's in.
The keyword `continue` skips the rest of the code in the loop.

Java has the same equalities/inequalities as Python, but not for logical operators:

Java	Python
&&	and
||	or
!	not

Java	Python

`if (boolean) {`
`  // Touch grass`
`} else if (boolean) {`
`  // Eat banana`
`} else {`
`  // Sleep`
`}`
	`if boolean:`
	`  // Touch grass`
	`elif Boolean:`
	`  // Eat banana`
	`else:`
	`  // Sleep`

`while (boolean) {`
`  // Touch grass`
`}`
 	
`do {`
`  // Touch grass`
`} while (Boolean);`
	`while boolean:`
	`  // Touch grass`
	
	`// Touch grass`
	`while boolean:`
	`  // Touch grass`

If/else branches with only one-liners in their body can omit {} to save space.
Here's a few Java/Python syntax comparisons on certain shortcut operations:

```java
int x = 1; int y = 2;
x += 1; x -= 1; x *= 1; x /= 1;

x = y = 0; // Python equivalent is x, y = 0, 0

boolean test = x < 3.5 && 3.5 < y;
String s = 5 > 2 ? "Yessir" : "Nope";
```

```python
x, y = 1, 2
x += 1
x -= 1
x *= 1
x /= 1
test = x < 3.5 < y
s = "Yessir" if 5 > 2 else "Nope"
```

Java does not have equivalents to list/set/dict comprehensions. It does however have special increments:

```java
int i = 3;
int a = i++;  // a == 3, i == 4
int b = ++a;  // a == 4, i == 4, b == 4
```

The **postincrement** `i++` increases `i` by 1 after a specified action (ie. setting it to `a`)
The **preincrement** `++a` increases `a` by 1 before a specified action (ie. setting it to `b`)

Java's for loop is more customizable than Python's. The termination/increment re-runs at each iteration; this can cause unintentional problems (eg. deleting list items while relying on `list.size()`).

```java
for (initiation; termination; increment) {
    // initiation: an expression that runs just as the loop starts. Optional.
    // termination: a boolean to stop at. Left blank, the loop runs forever.
    // increment: an expression that runs after every iteration. Optional.
}
```

Java

```java
int[] lst = {1, 2, 3, 4, 5};

for (int i = 0; i < lst.length; i++) {
    // Do something
}

int k = 1;
int j = 2;
for (++j; k + 2 >= -6;) {
    // Do something
    k -= 2;
}

for (int item : lst) {
    // Do something
}
```

Python

```python
lst = [1, 2, 3, 4, 5]

for i in range(len(lst)):
    // Do something


k, j = 1, 2

j += 1
while k + 2 >= -6:
    // Do something
    k -= 2

for item in lst:
    // Do something
```

Java also contains **switches**, which are similar to if/else branches, but I don't find them super useful.

```
String visiting = "Podgorica";
switch(visiting) {
    case "Windhoek": // Java 11 switches don't support multiple identical cases.
        System.out.println("Welkom in Namibië");
        break;  // If you don't put it in, system will print "Bem-vindo a Angola". Dunno why.
    case "Luanda":
        System.out.println("Bem-vindo a Angola");
        break;
    case "Skopje": case "Strumica":   // How to "overcome" the identical cases problem
        System.out.println("Dobredojdovte vo Severna Makedonija");
        break;
    default:   // What to do if there is no match
        System.out.println("Couldn't find the country you're visiting");
}
```

There's also the "enhanced switch", introduced in Java 13 with more features and is actually worth using (warning, Java 12's enhanced switch is slightly different); but CSC207 forces you to work in Java 11...

```
switch(visiting) {    // A switch statement.
    case "Asunción", "Ciudad del Este", "San Lorenzo", "Luque":
        System.out.println("Bienvenido a Paraguay");
    default:
        System.out.println("Selamat datang ke Brunei");
}
String favouritePokemon = "Porygon2";
String comment = switch(favouritePokemon) {     // A switch expression.
    case "Vaporeon" -> yield "Hey guys, did you know that in terms in male human and female…");
    case "Lopunny", "Braixen", "Lucario" -> yield "furry lol";
    case "Bruxish" -> yield "what is wrong with you";    // yield is like return, but it is
    default -> yield "Your taste is valid";              // usable only in switch expressions.
};  // The arrow -> is anonymous function notation (see below).
```

Anonymous Methods/Classes

Lambda functions are notated `param -> statement` or `(param1, param2, …) -> {statements}`, like so:

```
() -> System.out.println("Delta epsilon ptsd gang"); // No arguments!
(p1, p2) -> p1 + p2;                                 // Weird that Java doesn't enforce typing.
(int scarabs, String item) -> {                      // {} must surround multi-statements.
    int i = 1;
    scarabs >= 5 + i ? item : "You don't have enough scarabs!";   // Writing return is optional
}
```

Lambda functions can be multi-statement. Unlike in Python, Java uses lambda functions much more; you'll find them interspersed between sections. You can also create anonymous classes, like so:

```
System.out.println(new Object() {    // extends Object
    @Override
    public String toString() {
        return "Ba. Ha ba ba. Ha babada ga da. Ha baba da daba.";
    }
});  // Prints out "Ba. Ha ba ba. Ha babada ga da. Ha baba da daba."
```

Method Reference: A special, concise type of lambda expression of format `ClassName::funcName`. It's unfamiliar and just visual candy, so similarly like switches, it's not super important.

```
System.out::println                          (x) -> System.out.println(x)
coolClass::classMethod                       (coolClass x) -> x.classMethod()
Integer::compareTo                           (a, b) -> a.compareTo(b) // Generics!
```

There're also constructor references.

```
coolClass::new                                    () -> new classMethod()
// Depending on context, equivalent to the right: (x, y) -> new classMethod(x, y)
```

Inheritance

Here's a Java/Python comparison of inheritance. Abstract classes can inherit abstract classes, BTW.

```java
/**
 * Here's a Javadoc.
 * @param username something
 * @param password something else
 * @author me, cause I'm cool
 * @version 142857.1
 * @throws ExceptionName if it does
 */
abstract public class Parent {
    public String username = "Hungary";
    public String password = "Kolozsvar";

    public Parent() {
        System.out.println("Akmola");
    }

    public abstract void absMethod();

    public int notAbsMethod() {
        return 5;
    }
}
public class Child extends Parent {
    /* Here's a comment */
    public String username = "Romania";
    public String password = "Cluj-Napoca";
    public boolean isCool;

    public Child() {
        super();
        isCool = super.username != this.username;
        System.out.println("Qara Khitai");
    }

    public void absMethod() {
        // Implement the method.
    }

    @Override
    public int notAbsMethod() {
        return 10;
    }
}
```

```python
class Parent:
    """Here's a docstring.

    Instance Variables:
    -   username: something
    -   password: something else
    """
    username: str
    password: str

    def __init__(self) -> None:
        """Initiate the parent class."""
        self.username = "Hungary"
        self.password = "Kolozsvar"
        print("Akmola")

    def abs_method(self) -> None:
        """An abstract method."""
        raise NotImplementedError

    def not_abs_method(self) -> int:
        """A not abstract method."""
        return 5

class Child(Parent):
    # Here's a comment.
    is_cool: bool

    def __init__(self) -> None:
        """Initiate the child class."""
        Parent.__init__(self)
        self.username = "Romania"
        self.password = "Cluj-Napoca"
        self.is_cool = Parent.username != self.username
        print("Qara Khitai")

    def abs_method(self) -> None:
        """Implement the abstract method."""
        # Implement the method.

    def not_abs_method(self) -> int:
        """A not abstract method."""
        return 10
```

Abstract classes should be used when a method should be implemented differently in subclasses.
Abstract classes can extend other abstract classes. They can also have no abstract methods.

If no `super()` is written, constructors call `super()` with no arguments by default.
When a child class constructor calls `super()`, `super()`'s arguments must match the parent constructor.

Constructors are not obligatory in Java classes, unlike Python.
If no constructor is written, a **default constructor** inherited from Object will run, initializing class variables.

- If you create a constructor, you override the default constructor.
- Child classes will then need custom constructor too and call `super()`. Otherwise, no constructor will be run (since you overrode it) and there'll be an error.

Child constructors can have different arguments than parent constructors.

For overridden methods, it's good practice to write `@Override`: Java will check if you override properly. Also, you should write `this.` to distinguish the child variable from the parent variable.

If a child class contains no duplicate `var` or `method()` of a parent class, behaviour is the same as Python.

- `var` will access the parent variable
- `method()` will access the parent method

If a child class contains a duplicate `var` or `method()` of a parent class,

- `this.var` will access the child variable
- `this.method()` will access the child method
- `super.var` will access the parent variable
- `super.method()` will access the parent method

This is how the classes are instantiated.

```
Parent p = new Parent(); // Error -> Cannot instantiate an abstract class
Child c = new Child();
// "Akmola"
// "Qara Khitai"

c.username;        // "Romania"
c.password;        // "Cluj-Napoca"
c.notAbsMethod();  // 10
c.isCool();        // true
```

Casting

Casting: AKA typecasting, converting one data type into another. Created to solve problems in Java (that aren't in Python) that only exist because Java's so strict with types.

Upcasting: Casting a child class to the parent class.

```
Child c = new Child();
// "Akmola"
// "Qara Khitai"

((Parent) c).username;        // "Hungary"
((Parent) c).password;        // "Kolozsvar"
((Parent) c).notAbsMethod();  // 10 -> since the child method overrode the parent method
((Parent) c).isCool();        // error -> variable doesn't exist in parent class
```

Inside a class method, you may also do casting like `((Parent) this).varName`.

Downcasting: Casting a parent class to the child class.

```
((Child) ((Parent) c)).username;        // "Romania"
((Child) ((Parent) c)).password;        // "Cluj-Napoca"
((Child) ((Parent) c)).notAbsMethod();  // 10
((Child) ((Parent) c)).isCool();        // true
```

You can use downcasting as a "cheat" to store different data types on arrays.

```
Object[] stuff = new Object[5];  // Since everything inherits from Object, anything fits inside

Integer item1 = 88;
String item2 = "Hewwo";
Double[] item3 = {1.5, 3.0};  // Arrays inherit from Object too!

stuff[0] = item1; stuff[1] = item2; stuff[2] = item3; stuff[3] = 5; // Autoboxing for stuff[3]!
```

This works as arrays can contain instances of child classes. Integer, String, and Float all inherit from Object.

However, working with this list can be challenging.

```
int indexZero = stuff[0]            // Error! Java only knows stuff[0] is an Object.
int indexZero = (int) stuff[0]      // Downcast it to an int
Object indexZero = stuff[0]         // Also acceptable, since you've defined stuff as Object[]

// What if you want to iterate through the array?
for (Object item : stuff) {
    // stuff is defined as Object[], so you're forced to say item is an Object too.

    // It's dangerous to cast inside a loop, as not all items are guaranteed to be castable.

    /* If you want to check the types of each item…
    Java equivalent of Python's type():  var.getClass().getName()
    Or, if you want a boolean:           var instanceof String
    */
}
```

Widening Casting: Memory-wise, casting a smaller type to a larger type. Done mostly automatically.

```
// byte -> short -> int -> long -> float -> double
//          char -> int
short a = 1;
long b = a;     // AKA long b = (long) a
double c = b;   // AKA double c = (double) b

a;    // 1
b;    // 1
c;    // 1.0

byte w = 97;
char x = w;      // Error, can't automatically cast numbers to char
char x = (x) w;  // For char, you must manually cast
int y = x;
float z = x;

x;                  // 'a'  -> think of it like Python's chr() and ord()
y;                  // 97
z;                  // 97.0
x == z & y == z;    // true
```

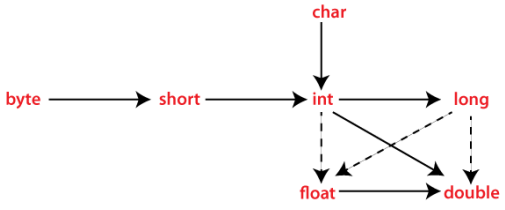
Narrowing Casting: Memory-wise, casting a larger type to a smaller type. Must be done manually.

```
// double -> float -> long -> int -> char -> short -> byte
double a = 1286.942012345678912 34;
float b = (float) a;  // If you do not include (float), there'll be a type error
long c = (long) b;
byte d = (byte) c;

a;                          // 1286.942012345679 -> dropping decimals
b;                          // 1286.942 -> even more dropped decimals
c;                          // 1286 -> takes the floor of 1286.942
d;                          // 6 -> returns 1286 % 128
a == b || b == c || c == d;  // false
```

The graphic on the right summarizes the results of type coercion in Java.

I've heard from online sources that you should generally avoid casting and checking types whenever possible, because it can be indicative of poor design choices that can be redesigned better.



Interfaces

Interface: A special type of abstract class specifying behaviours that must be implemented.

Implementing an Interface	Extending a Class
A common functionality that a bunch of classes happen to have	A top-down tree hierarchy between a parent class and many children
Implementers do not have to be related, apart from the common functionality	Children have all parent features, and are closely-related to the parent
Makes implementations stable/secure by defining a protocol of behaviours	The basic template for inheritance.
eg. Lists, Sets, and Maps all implement Iterable, Collection, and Cloneable	eg. LinkedList, ArrayList, Array are all children of abstract class List

Classes can `implement` interfaces and `extend` classes. They cannot `extend` interfaces.
Interfaces can `extend` interfaces. They cannot `implement` interfaces or `extend` classes.

Interfaces must be `public` and have no constructors.
Interfaces can only have `public static final` variables and `public abstract` methods.

Multiple Inheritance When child classes can inherit from multiple parent classes. Unlike Python, Java's `extend` doesn't support multiple inheritance (thanks, Java), but `implements` supports multiple interfaces.

```
public interface Corrosive {
    boolean is_corrosive = true; // This is callable in any implementer

    void corrode();       // Must be implemented

    String getWarning();  // Must be implemented
}

public class SodiumHydroxide implements Corrosive, WaterSoluble {
    // Pretend there's another interface called WaterSoluble
    public void corrode() {
        System.out.println("Ouchy ouchy, OH- is hurty!!!");          // Method implemented!
    }

    public void dissolve() {
        System.out.println("Your water now contains some basic OH-.");
    }

    public String getWarning() {                                    // Method implemented!
        return "Sodium hydroxide can cause severe blisters/caustic burns/corrosion
            to the skin and as well as permanent eye damage.";
    }
}

public class AlconoxDetergent extends Detergent implements Corrosive {
    // Pretend there's another class called Detergent
    public void corrode() {                                         // Method implemented!
        System.out.println("Ouchy ouchy, detergent is corrody!");
    }

    public String getWarning() {                                    // Method implemented!
        return "This product is corrosive; exposure can irritate eyes, respiratory
            system, and skin, and cause burns on contact.";
    }
}
```

Polymorphism

Polymorphism: A feature of object-oriented programming where methods can work on very generic inputs – arrays of any objects, parent and child classes, implementers of an interface, etc.

Shadowing: When the variables/methods of a parent class replace that of the child's
Overriding: When the variables/methods of a child class replace that of the parent's

We can set objects to an instance of their child class (but not vice versa). Parent can also be abstract.

```
Parent p = new Child();
```

If the child class contains a duplicate `var` or `method()` as the parent class,

- `var` will access the parent variable (shadow)
- `method()` will access the parent method (shadow) if `method()` is static
- `method()` will access the child method (override) if `method()` is not static

Upcasting has no effect. `((Parent) p).var` will follow the same behaviour as `p.var`, for instance.
Downcasting will have effect. `((Child) p).var` will not follow the same behaviour as `p.var`, for instance.

Here's an example of polymorphism between parent/child classes and interfaces:

```
interface Math {
    void eval();
}
interface Greek {
    void pronounce();
}
class Letter {
    public void italicize() {}
}

class X extends Letter implements Math {…} // I'm going to omit all the implementations
class Limit implements Math {…}
class Delta extends Letter implements Math, Greek {…}
class Epsilon extends Letter implements Math, Greek {…}

Math expr1 = new X();
Limit expr2 = new Limit();
Greek expr3 = new Delta();
Letter expr4 = new X();
Math[] exprArray = {expr1, expr2, (Math) expr3, (Math) expr4, new Epsilon()};
// expr2 is a Limit, which implements Math. It doesn't need to be casted.
// expr3 is a Greek, but since it's set to a Delta which also implements Math, it can be casted
// expr4 is a Letter, but since it's set to a X which also implements Math, it can be casted
// Since the Epsilon class implements Math, it doesn't need to be casted.

expr3 instanceof Letter && expr3 instanceof Math && expr3 instanceof Greek; // true

expr1.italicize();   // Error -> expr1 being saved as Math limits it to eval()
expr1.eval();
Letter test1 = (Letter) expr1;
test1.italicize();
test1.eval();       // Error -> test1 being saved as Letter limits it to italicize()

expr3.eval();       // Error -> expr3 being saved as Greek limits it to pronounce()
expr3.italicize();  // Error -> expr3 being saved as Greek limits it to pronounce()
expr3.pronounce();
Math test2 = (Delta) expr3;
test2.eval();
test2.italicize();  // Error -> test2 being saved as Math limits it to eval()
test2.pronounce();  // Error -> test2 being saved as Math limits it to eval()
```

Here's an example of polymorphic functions in action, following good design principles.

```
public abstract class Shape {
    public abstract double area();   // All subclasses of shape must have area() implemented
}

public class Circle extends Shape {
    private double radius = 1;
    private final double pi = 3.14159265;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double area() {
        return radius * radius * pi;  // pi * r^2
    }
}

public class Triangle extends Shape {
    private double base = 1;
    private double height = 1;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    public double area() {
        return base * height * 0.5;   // bh/2
    }
}

public class AreaCalculator {
    public static double[] area(Shape[] shapes) {     // Works on all subclasses of Shape
        double[] areas = new double[shapes.length];
        for (int i = 0; i < shapes.length; i++) {
            areas[i] = shapes[i].area();
        }
        return areas;
    }
}

public class Starter {
    public static void main() {
        Shape s1 = new Circle(5);           // Shape is abstract, but it's child Circle isn't,
        Shape s2 = new Triangle(21, 3);     // so that is okay.
        Shape s3 = new Triangle(2.1, 4);
        Shape s4 = new Circle(2.5);
        Shape[] s = {s1, s2, s3, s4};
        double[] areas = AreaCalculator.area(s);  // Area is calculated uniquely for each shape

        // areas is a double[] consisting of [78.53981625, 31.5, 4.2, 19.6349540625]
    }
}
```

Open/Closed Principle: Code should be open to extension and closed for modification.

- We can easily extend this program by adding more subclasses of shapes
- We do not need to modify existing shapes (or the area calculator) in the process

Generics

Generics: Annotations that enforce data type restrictions.
They're notated `< >`; only reference types and classes are allowed inside these brackets.

ArrayLists are **parameterized classes**; they use generics. They behave like `Object[]` without generics.
In fact, ArrayLists without generics are flagged in IntelliJ: "Raw use of parameterized class 'ArrayList'"

```
// new ArrayList<String>() -> empty Arraylist of strings
// new ArrayList<Integer>(4) -> Arraylist of length 4 with only 0s (default integer value)

String[] places1 = {"Emirate of Bukhara", "Sami Republic", "Chukchi Sea"};

ArrayList places2 = new ArrayList();   // Warning: "Raw use of parameterized class"
// This is equivalent to ArrayList<Object> places2 = …

places2.add("Sakhalin Island");  // Acceptable, but IntelliJ will warn of unchecked types
places2.add(69);                 // Acceptable, but IntelliJ will warn of unchecked types.
places2.add(places1[0]);         // Acceptable, but IntelliJ will warn of unchecked types.

String item = places2[0];           // Error! Java only knows places2's items are Objects.
String item = (String) places2[0];  // Acceptable after downcasting

// places2 is an ArrayList<Object>, ["Sakhalin Island", 69, "Emirate of Bukhara"]
```

To make ArrayLists behave like `String[]` or `int[]`, ArrayLists need to specify generics. There're many notation pitfalls and errors.

```
ArrayList<> places3 = new ArrayList();
// Error, we don't know what <> refers to.

ArrayList<String> places3 = new ArrayList<Object>();
// Can't set Array<ChildClass> to an instance of Array<ParentClass>, only the other way around.
```

The following are "acceptable", but IntelliJ nevertheless gives warnings:

```
ArrayList<String> places3 = new ArrayList();
// Java "doesn't know" the type of the new ArrayList(), so IntelliJ gives a warning.

ArrayList<String> places3 = new ArrayList<String>();
// This is redundant; too verbose.
```

The following code is the proper way to make ArrayList behave like a mutable `String[]`.

```
ArrayList<String> places3 = new ArrayList<>(Arrays.asList(places1));
// ArrayList(Arrays.asList(an array)) initiates an ArrayList made of all elements in the array.
// List.of("a", "b", "c", "d") is similar to Arrays.asList("a", "b", "c", "d")
// Both initializes a new List, like Python's … = ["a", "b", "c", "d"]

places3.add("Kolyma River");
places3.add(404);             // Error! Only strings are allowed in places3.
places3.add(places1[0]);

Object item = places3[0];     // Acceptable.
String item = places3[0];     // Also Acceptable.

// places3 is an ArrayList<String>, ["Emirate of Bukhara", "Sami Republic" , "Chukchi Sea",
                                     "Kolyma River", "Emirate of Bukhara"]
```

`List.of` creates an immutable list that cannot handle `null`. It's slightly faster and takes less memory space.
`Arrays.asList` creates a mutable list that can handle `null` and implicitly calls `new`.

I don't know if this matters too much, since you're converting the `List` to an `ArrayList`.

Generics can also be used to create polymorphic methods.

Right before the return type, `<GenericName>` is written. It can be any name, but there're conventions:

- `<E>`: An element (of a collection)
- `<K>`: A key (of a map)
- `<V>`: A value (of a map)
- `<N>`: A number
- `<T>`, `<U>`, `<S>`: A data type
- `<X>`: An exception

References are then made to `GenericName` in the function, which enforce consistency in typing.

```
/** Mutate collection by appending onto it everything in items.
 * @param items an array of type T
 * @param collection a collection of type T
 * @return nothing, as this is a mutating function
 */
public static <T> void fromArrayToCollection(T[] items, Collection<T> collection) {
    for (T item : items) {
        collection.add(item);  // Collection is an interface. It has the method add()
    }
}

Integer[] numArray = {1, 2, 3};
List<Integer> numArrayList = new ArrayList<>();
fromArrayToCollection(numArray, numArrayList);  // T is replaced with Integer
// numArrayList is an ArrayList<Integer>, [1, 2, 3]

String[] strArray = {"a", "b", "c"};
List<String> strArrayList = new ArrayList<>();
fromArrayToCollection(strArray, strArrayList);  // T is replaced with String
// strArrayList is an ArrayList<String>, ["a", "b", "c"]
```

Without generics, to imitate the above, you'd have to overload a function with repetitive arguments:

- `Integer[]` items and `Collection<Integer>` collection
- `String[]` items and `Collection<String>` collection

But with generics, as long as the collection and array contain the same type, the function will substitute `GenericName` for that type and it will work.

Special Method Implementations

When overriding `.equals()`, implementations should satisfy the following for non-null references `a` and `b`:

1) **Symmetry:** `a.equals(b)` ⇔ `b.equals(a)`
2) **Reflexivity:** `a.equals(a)`
3) **Transitivity:** `a.equals(b)` and `b.equals(c)` ⇒ `c.equals(a)`

Also consider overriding `.hashCode()`, which returns an `int`, as `a.equals(b)` ⇒ `a.hashCode() == b.hashCode()`
But do note it's not a biconditional. I also don't know how `.hashCode()` works lol.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)      // A useful case to add to prevent NullPointerExceptions
        return false;
    if (getClass() != obj.getClass())
        return false;     // Another useful case to exclude different classes
    …                     // We can now assume obj is the same class, and downcast it.
}
```

There are two ways to create ways to compare classes.

One is to implement interface `Comparable<Type>` with the method `public int compareTo(Type objName)`.

```
class CoolBeans implements Comparable<CoolBeans> {
    // If you just wrote Comparable, you'd have to make CoolBeans comparable with all Objects.
    private String bean;

    public CoolBeans(String bean) {
        this.bean = bean;
    }

    /** Because compareTo should decide <, =, or >, a boolean won't work here.
     * Instead, Java makes you return an integer.
     * If the integer == 0, both objects are equal.
     * If the integer > 0, the other object is bigger.
     * If the integer < 0, the other object is smaller.
     */
    public int compareTo(CoolBeans other) {
        if (other.bean.length() > this.bean.length())
            return 971349;   // Any positive integer works! But I'd recommend just returning 1
        if (other.bean.length() < this.bean.length())
            return -1;
        return 0;
    }
}
```

Unlike in Python, implementing this won't let you use the inequality symbols (<, >, <=, >=).

```
// The closest equivalent to >=
boolean greaterOrEqual = Something.compareTo(Something) >= 0 ? true : false;

// Implementing Comparable does provide a shortcut in Arrays.sort() and similar sorting methods
CoolBeans item1 = new CoolBeans("Jet fuel can't melt steel beams, sus");
CoolBeans item2 = new CoolBeans("Among drip sus sussy");
CoolBeans item3 = new CoolBeans("https://youtu.be/oHg5SJYRHA0");
CoolBeans[] arr = {item1, item2, item3};

Arrays.sort(arr);      // arr is an array of CoolBeans, [item1, item3, item2]
```

Alternatively, to compare objects by multiple metrics, we create a class to implement `Comparator<Type>` with the method `public int compare(Type obj1, Type obj2)`.

```
class SusComparator implements Comparator<CoolBeans> {
    /** This works similarly to compareTo(), where you return an integer.
     * If the integer == 0, both objects are equal.
     * If the integer > 0, obj1 is bigger than obj2.
     * If the integer < 0, obj1 is smaller than obj2.
     */
    public int compare(CoolBeans bean1, CoolBeans bean2) {
        // To get a Java equivalent of str.count(substr), you have to go around importing,
        // so instead, consider this one-liner I found on StackOverflow:
        int sus1 = (bean1.length() - bean1.replace("sus", "").length()) / "sus".length();
        int sus2 = (bean2.length() - bean2.replace("sus", "").length()) / "sus".length();

        if (sus1 == sus2)
            return 0;
        return sus1 > sus2 ? 1 : -1;
    }
}
```

We can pass in an additional argument to `Arrays.sort()` and sort by a different comparator!

```
Arrays.sort(arr, SusComparator);            // arr is CoolBeans[], [item2, item1, item3]
Arrays.sort(arr, SusComparator.reversed()); // arr is CoolBeans[], [item3, item1, item2]
// All comparators have a .reversed() method which reverses their natural order.
```

Errors and Exceptions

Error: Serious, abnormal problems that a reasonable program should not try to work around.

- eg. `AssertionError, OutOfMemoryError`

Exception: Problems that a reasonable program might want to work around.

- **Checked Exception:** Exceptions found when Java is compiling your project
  - eg. `IOException, FileNotFoundException`
- **Unchecked Exception:** Subclasses of `RuntimeException`; exceptions found while code is running
  - eg. `ArithmeticException, NullPointerException`

Exceptions are Objects. Compare the following Java/Python code.

```java
public class NameError extends Exception {
    public NameError (String message) {
        super(message);
    }
}
// Your Javadoc should say @throws NameError if …
class City {
    static void checkName(String city) throws NameError {
        if (city.equals("Kiev")) {
            throw new NameError("It's Kyiv, not Kiev");
        } else if (5 / 0 == 10) {
            throw new NameError("It's İzmir, not Smyrna");
        }
    }
}
class Starter {
    public static void main(String[] args) {
        try {
            City.checkName("Kiev");
            City.checkName("Semipalatinsk");
        } catch (NameError e) {
            System.out.println(e);
        } catch (ArithmeticException e) {
            e.printStackTrace();  // Prints error location
        } finally {
            System.out.println("Semey");
        }
        // NameError: It's Kyiv, not Kiev
        // Semey
    }
}
/* If the line City.checkName("Kiev") is omitted, you get:
java.lang.ArithmeticException: / by zero
        at City.checkName(City.java:5)
        at Starter.main(Starter.java:6)
Semey */
```

```python
class NameError(Exception):
    """For wrong city names."""
    def __init__(self, message: str):
        """Initialize the exception."""
        Exception.__init__(message)


def checkName(city: str) -> None:
    """Check if the city name is right."""
    if city == "Kiev":
        raise NameError("It's Kyiv, not Kiev")
    elif 5 / 0 == 10:
        raise NameError("It's İzmir, not Smyrna")



if __name__ == "__main__":
    try:
        checkName("Kiev")
        checkName("Semipalantinsk")
    except NameError:
        print("NameError: It's Kyiv, not Kiev")
    except ArithmeticException:
        print("ArithmeticException")
    finally:
        print("Semey")

    # If no error is specified in "except:",
    # any exception will trigger it.

    # Python has an extra "else:" which runs if
    # no exception is reached
```

Catching an Exception will also catch all subclasses of the Exception. You shouldn't catch Errors.

The parent class to all errors is `Error`, and to all Exceptions is `Exception` (you can throw these!)
The parent class to `Error` and `Exception` is `Throwable`.

`Throwable` can be initialized as `new Throwable()` or `new Throwable("Error Message")`. You can use getMessage() to return the error message as a string.

You can throw pre-existing Exceptions and add custom string messages in the constructor.

Java programs end with exit codes. Exit codes of 0 mean no-error exits; 1/-1 means error-induced exits.

Iterators

Iterator: A class meant for iterating. Iterators can safely delete objects while iterating, unlike traditional element-based loops, which can't and raises a `ConcurrentModificationException`, or index-based loops, which rely on `list.size()`, which changes each time an item is deleted. Thanks, Java.

```
ArrayList<String> anime = new ArrayList<>(Arrays.asList("Pingu", "Cory", "Pop Team Epic"));

Iterator<String> iterator1 = anime.iterator();   // All Iterables have the .iterator() method.
iterator1.forEachRemaining((item) -> System.out.println(item));   // A 1-liner for-loop!
// Prints every item in anime
// forEachRemaining() only takes in anonymous functions
iterator1.forEachRemaining(System.out::println);
// Prints nothing, because iterator1 has already iterated to the end.
// forEachRemaining() only iterates through remaining items in the iterator.

Iterator<String> iterator2 = anime.iterator();
while (iterator2.hasNext()) {
    System.out.println(iterator2.next());  // Print the current item
}

for (Iterator<String> iterator3 = anime.iterator(); iterator.hasNext();) {
    iterator3.next();        // Must be called before a .remove()
    iterator3.remove();      // Deletes the just-iterated-over item from the list.
    // iterator.remove() only works for collections that allow deletion.
}
```

ListIterator: A special child class of Iterator, it works only on lists unlike Iterator.

Method	Description
`add(E element)`	Insert element at the current index
`remove()`	Remove the previously-iterated-over item
`set(E element)`	Set the previously-iterated-over item to element
`hasNext()`	If this item isn't the last item
`next()`	Go to next index, return the item there
`nextIndex()`	Return the index of the next list item
`hasPrevious()`	If this item isn't the first item
`previous()`	Go to previous index, return the item there
`previousIndex()`	Return the index of the previous list item

Iterables also have `.forEach()`, similar to `.forEachRemaining()` but always iterating from start to end:

```
ArrayList<String> lines = new ArrayList<>(Arrays.asList("Curzon", "Maginot", "Bar Lev"));

lines.forEach((item) -> System.out.println(item));
// Prints every item in lines. Very clean 1-liner!
```

Iterating through sets is similar to lists, but maps are a little more difficult.

```
HashMap<String, Double> atomicMass = new HashMap<>();
atomicMass.put("Thulium", 168.93422);
atomicMass.put("Molybdenum", 95.95);
atomicMass.put("Oganesson", 295.216);

// You can do a regular for loop through atomicMass.keys() and atomicMass.values().
// But if you want to access a key and its value, it's faster to use another method:

for (Map.Entry<String, Double> entry : atomicMass.entrySet()) {   // All maps have .entrySet()
    String k = entry.getKey();
    double v = entry.getValue();
    // Do something with the key-value pair
}
```

```java
// Another method: iterators!
Iterator<Map.Entry<String, Double>> iterator = atomicMass.entrySet().iterator();
iterator.forEachRemaining((entry) -> {   // Loop using forEachRemaining
    System.out.println(entry.getKey() + " has an atomic mass of " + entry.getValue());
    // Thulium has an atomic mass of 168.93422
    // … (order is not guaranteed since it's a hashmap)
});
while (iterator.hasNext()) {            // Traditional while loop using hasNext()
    Map.Entry<String, Double> entry = iterator.next();
    String k = entry.getKey();
    double v = entry.getValue();
    // Do something with the key-value pair
}

// Another method: forEach()
atomicMass.forEach((k, v) -> System.out.println(k + " has an atomic mass of " + v));
```

Optionals

Optionals: A parameterized class very similar to Python's `Optional[…]` type annotation.

```java
Optional<String> cycle1 = "Calvin";                // Error, an Optional isn't a String
Optional<String> cycle2 = Optional.of("Krebs");    // Correct
Optional cycle3;                                   // "Raw use of parameterized class" warning
Optional<String> cycle4 = null;                    // Acceptable, but not recommended
Optional<String> cycle5 = Optional.empty();        // Preferred

// ofNullable() can take both input types
Optional<String> cycle6 = Optional.ofNullable("Value");
Optional<String> cycle7 = Optional.ofNullable(null);

cycle2.isPresent();  // true
cycle3.isPresent();  // error, cycle3 has not been initialized
cycle5.isPresent();  // false

cycle2.orElse(5);             // Krebs      - return original value if it exists
cycle5.orElse("Glyoxylate");  // Glyoxylate – return "else" value if no original value exists

cycle2.ifPresent((x) -> {     // Equivalent to if (x.isPresent()) {…}
    System.out.println("Oxaloacetates are a cool part of the " + x + "cycle");
    System.out.println("Wow");
});
```

There also exists the `OptionalInt`, `OptionalDouble`, and `OptionalLong` classes, which, for some reason, don't implement/extend the `Optional` class. Why those three specifically? Dunno.

```java
// You can also use all the above Optional methods: .empty(), .of(), .orElse(), isPresent()
OptionalInt n1 = OptionalInt.of(1);
OptionalLong n2 = OptionalLong.empty();

n1.getAsInt();     // 1
n2.getAsLong();    // Throws NoSuchElementException
```

Streams

Streams: A parameterized class that brings functional programming to Java. It contains/processes sequence of items and supports some convenient features (that Python has but regular Java doesn't). Stored in `java.util.stream`. Stream methods usually don't mutate, and are often chained together.

- **Intermediate Operation:** Methods that filter/map/sort and return a stream
- **Terminal Operations:** Methods that collect/reduce/return a non-stream type

Java	Python
`import java.util.stream.*;`	
`// Let's first work with IntStream, a child class of Stream` `// There's also LongStream and DoubleStream`	
`int[] ints = {1, 2, 1, 3, 4, 512, -1, 0, -2, -3, 2, 1, -10};`	`ints = [1, 2, 1, 3, 4, 512, -1, 0, -2, -3, 2, 1, -10]`
`// Create/modify an IntStream`	
`IntStream.of(ints);  // the stream has ints' values`	
`IntStream.of(1, 2, 3, 4, ints);`	`[1, 2, 3, 4] + ints`
`IntStream.range(1, 10);  // goes from 1 to 9`	`range(1, 10)`
`IntStream.rangeClosed(1, 10);  // goes from 1 to 10`	`range(1, 11)`
`IntStream.of(ints).limit(10);  // filters out items > 10`	`[x for x in ints if x > 10]`
`// IntStream-specific methods that terminate`	
`IntStream.of(ints).sum();  // int, 510`	`sum(ints)`
`IntStream.of(ints).max();  // OptionalInt, 512`	`max(ints)`
`IntStream.of(ints).min();  // OptionalInt, -10`	`min(ints)`
`IntStream.of(ints).average();  // OptionalDouble, 39.23076…`	`sum(ints) / len(ints)`
`IntStream.of(ints).sorted();  // sort the IntStream`	`sorted(ints), sort(ints)`
`IntStream.of(ints).summaryStatistics();` `// Returns the class IntSummaryStatistics with methods methods getMax/Average/Count/Min/Sum()`	`# No equivalent`
`// Generic Stream methods`	
`IntStream.of(ints).distinct();  // filters out repetitions`	`list(set(ints))`
`IntStream.of(ints).skip(3);  // discards first 3 elements`	`ints[3:]`
`IntStream.of(ints).count();  // long, 13`	`len(ints)`
`IntStream.of(ints).toArray();  // int[], the original ints`	
`IntStream.of(ints).findFirst();  // OptionalInt, 1`	`ints[0]`
`// In IntStreams, sorted(), max(), and min() optionally take Comparators. In generic Streams, Comparators are mandatory.`	

Here are some more generic stream methods:

Java	Python
`String[] rou = {"Beijing kao ya", "Ji ding", "Chashao rou"};`	`rou = ["Beijing kao ya","Ji ding", "Chashao rou"]`
`ArrayList<String> buShiRou = new ArrayList<>(Arrays.asList("Biangbiang mian", "Cong you bing", "You tiao", "Zhajiang mian"));`	`buShiRou = ["Biangbiang mian", "Cong you bing", "You tiao", "Zhajiang mian"]`
`Arrays.stream(rou);  // Creates Stream via an array`	
`buShiRou.stream();  // Creates Stream via non-array`	
`Stream.concat(Arrays.stream(rou), buShiRou.stream());`	`rou + buShiRou`
`buShiRou.stream().allMatch(x -> x.contains("mian"));  // false`	`all("mian" in x for x in buShiRou)`
`buShiRou.stream().anyMatch(x -> x.length() > 10));  // true`	`any(len(x) > 10 for x in buShiRou)`
`buShiRou.stream().noneMatch(x -> true);  // false`	`not any(true for x in buShiRou)`
`buShiRou.stream().filter(x -> x.startsWith("You"));`	`[x for x in buShiRou if x[:3] == "You"]`
`buShiRou.stream().map(String::toLowerCase);`	`[x.lower() for x in buShiRou]`
`buShiRou.stream().flatMap(x -> Stream.of(x, x + "!"));`	`flatten([[x, x + "!"] for x in buShiRou])`
`// You can convert between types with mapToInt/Long/Double`	`# Where flatten([[a], [b]]) == [a, b]`

`buShiRou.stream().reduce("Leng mian", (a, b) -> a + " and " + b));`	`a = "Leng mian: "`
	`for b in buShiRou:`
	`  a = a + " and " + b`
	`return a`

`/* Returns the string "Leng mian and Biangbiang mian and Cong you bing and You tiao and Zhajiang mian" */`	
`buShiRou.stream().forEach(System.out::println);`	`for x in buShiRou:` `  print(x)`

The Stream method, `collect()`, uses a class called `Collector` with many static methods:

Java

```java
Stream<String> s = Stream.of("Bauhaus", "Brutalist", "Art
Deco", "Cape Cod", "Moorish");

s.collect(Collectors.toSet());
s.collect(Collectors.toList());
s.collect(Collectors.toMap(x -> x, x -> x.length()));
// You can write Function.identity() to replace x -> x

s.collect(Collectors.toMap(x -> x.length(), x -> x);
// Error – duplicate keys detected!

s.collect(Collectors.toMap(x -> x.length(), // Keys
                           x -> x,          // Values
/* Duplicate keys case */  (v1, v2) -> v1 + ", " + v2,
/* Tree type of result */  TreeMap::new));

/* Result:
{7=Bauhaus, Moorish, 8=Art Deco, Cape Cod, 9=Brutalist}
*/

s.collect(Collectors.groupingBy(x -> x.length()));
/* Returns a HashMap,
{7=[Bauhaus, Moorish], 8=[Art Deco, Cape Cod],
9=[Brutalist]} */

s.collect(Collectors.partitioningBy(x -> x.contains(" ")));
/* Returns a HashMap, {false=[Bauhaus, Brutalist,
Moorish], true=[Art Deco, Cape Cod]} */
s.collect(Collectors.partitioningBy(x -> x.contains(" "),
/* Optional 2nd arg, type */      Collectors.toSet()));

s.collect(Collectors.joining("|"));
s.collect(Collectors.summingInt(String::length));
s.collect(Collectors.averagingInt(String::length));
// summing/averaging also have Double/Long versions,
// which take lambda functions that return doubles/longs

// collect() allows multiple Collector function arguments
// like: s.collect(Collectors.…(), Collectors.…(), …)
```

Python

```python
s = ["Bauhaus", "Brutalist", "Art Deco",
"Cape Cod", "Moorish"]

set(s)
list(s)
{x: len(x) for x in s}


{len(x): x for x in s}
# No error; earlier keys are overwritten

d = {}
for x in s:
    if len(x) in d:
        d[len(x)] = d[len(x)] + ", " + x
    else:
        d[len(x)] = x

d = {}
for x in s:
    if len(x) in d:
        d[len(x)].append(x)
    else:
        d[len(x)] = [x]

d = {False: [x for x in s if " " not in s],
     True: [x for x in s if " " in s]}

d = {False: {x for x in s if " " not in s},
     True: {x for x in s if " " in s}}

"|".join(s)
sum(len(x) for x in s)
sum(len(x) for x in s) / len(s)
```

Collectors also has the functions `mapping()`, `flatMapping()`, and `reducing()`, which work just the like equivalent Stream methods, so I don't see the point of them.

Records

Records: A restricted class meant to be a "plain data carrier", immutable, with only basic methods like constructors and getters. This was introduced in Java 14 concept, so skip this if you're in CSC207.

```java
public final class Tuple {
    private final List<String> contents;
    // Constructor for contents
    // Getters for contents
}
```

```java
public record Tuple(List<String> contents) {
    // You can add more methods
}

// Access contents with Tuple.contents()
```

The left and right sides are equivalent. You can add extra methods in the record if you want. Anyways, this is really just a shortcut class, so it's not super important to learn.

Nested Classes and Methods

For the sake of readability/encapsulation/grouping similar classes, you can nest classes. The following code is garbage (don't design your program like this!), but it shows the properties of nested classes well.

```
class Alkane extends Hydrocarbon {
    private int c;
    private String IUPAC;

    public Alkane(int c) {
        this.c = c;
        IUPAC = Nomenclature.prefix.get(c) + "ane"; // Can access private variables
    }
    public Alkane(String IUPAC) {
        this.IUPAC = IUPAC;
        this.c = Nomenclature.getCarbonsFromIUPAC(IUPAC); // Can access private methods
    }
    public int getCarbons() { return c; }

    public double calculateMass() {
        HelperMethods hi = new HelperMethods();
        return hi.calculateMass();
        // You can alternatively write Alkane.HelperMethods.calculateMass(…)
    }

    public static class Nomenclature {
        // Private methods/variables in here can be accessed from anywhere in the class Alkane
        private static final HashMap<Integer, String> prefix = Map.of(1, "Meth", 2, "Eth", 3,
        "Prop", 4, "But", 5, "Pent", 6, "Hex", 7, "Hept", 8, "Oct", 9, "Non", 10, "Dec");
        // Use List.of(), Set.of(), and Map.of() to initialize collections in 1 line.

        private int x = 1;
        /* In static classes, non-static methods can access non-static vars/methods only if
        both are in the same class. */
        public int getX() { return x; }
        /* A non-static method outside Nomenclature can access a non-static method/var in
        Nomenclature only if an instance of Nomenclature is created */

        // Static methods can only access static variables/methods.
        public static String getFormula(int c) { return "C" + c + "H" + 2 * c + 2;}
        // Even if getFormula() is not static, it still can't access c from Alkane

        private static String getCarbonsFromIUPAC() {…}
        public String getIUPAC() {…}
    }

    // Classes containing static methods must be labelled static classes
    private class HelperMethods {
        public final double avogadro = 6.0221407621e23;

        public double calculateMass() { return (c * 12 + (2 * c + 2)) / avogadro; }
        // You can access c from the class Alkane
    }
}
/* Alkane.Nomenclature.getFormula() is the only directly-callable method in Nomenclature

You must do "Alkane.Nomenclature hi = new Alkane.Nomenclature()" before being able to access
hi.getIUPAC() and hi.getX()

No methods in HelperMethods are callable, even public ones, as HelperMethods is a private class
*/
```

Static Blocks

Static Block: A special static code chunk inside a class that *runs exactly once*, at the first time the class is instantiated or a class method is called. Doesn't serve much useful purpose, to be honest.

```
public class Quotes {
    static int i = 1;
    static {
        i = 0;
        System.out.println("Neia teia an da ka");
    }
    // You can't access non-static variables or
    // methods in the static block
}
```

```
// Somewhere else…
System.out.println(Quotes.i);
// 0
// Neia teia an da ka
System.out.println(Quotes.i);
// 0

// No point in setting i = 1, as the static
// block sets i = 0. Instantiating Quotes
// will not reset i = 1 since i is static
// (i = 1 runs only when the code starts)
```

Enums

Enum: A type of class that stores predefined constants.

```
public enum PlaylistToRelaxTo {
    STRAVINSKY_RITE_OF_SPRING, RAUTAVAARA_PIANO_CONCERTO_1, LIGETI_ATMOSPHERES, LISZT_TOTENTANZ,
    PENDERICKI_THRENODY, GUBAIDULINA_DE_PROFUNDIS, XENAKIS_MISTS; // Constants are UPPPER_CASE
} // Switches or if/else statements are commonly used with enums.

PlaylistToRelaxTo.PENDERICKI_THRENODY;  // return PENDERICKI_THRENODY of class PlaylistoRelaxTo
PlaylistToRelaxTo.values();             // returns an array of enum values

for (PlaylistToRelaxTo p : PlaylistToRelaxTo.values()) {
    System.out.print(p.name().replace('_', ' ').toLowerCase() + ", ");
}   // Prints "stravinsky rite of spring, rautavaara piano concerto 1, ligeti atmospheres, …"
```

Enums can store more than just a single value, however.

```
public enum AlkanMinorEtudes {
    COMME_LE_VENT            ("A", "Prestissimamente", 4, 35),
    EN_RYTHME_MOLOSSIQUE     ("D", "Risoluto", 8, 14),
    SCHERZO_DIABOLICO        ("G", "Prestissimo", 4, 32),
    SYMPHONY_FOR_SOLO_PIANO_1 ("C", "Allegro moderato", 10, 43),
    SYMPHONY_FOR_SOLO_PIANO_2 ("F", "Marche funèbre (Andantino)", 8, 15),
    SYMPHONY_FOR_SOLO_PIANO_3 ("Bb", "Menuet (Tempo di minuetto)", 5, 42),
    SYMPHONY_FOR_SOLO_PIANO_4 ("Eb", "Finale (Presto)", 4, 36), // BTW I recommend this piece
    CONCERTO_FOR_SOLO_PIANO_1 ("G#", "Allegro assai", 29, 34),
    CONCERTO_FOR_SOLO_PIANO_2 ("C#", "Adagio", 12, 6),
    CONCERTO_FOR_SOLO_PIANO_3 ("F#", "Allegretto alla barbaresca", 10, 17),
    OVERTURE                 ("B", "Maestoso-Lentement-Allegro", 15, 46),
    LE_FESTIN_D_ESOPE        ("E", "Allegretto senza licenza quantunque", 9, 4);

    public final String key; // Define the variables used above
    public final String tempo;
    public final int lenMins;
    public final int lenSecs;

    AlkanMinorEtudes(String key, String tempo, int lenMins, int lenSecs) {
        this.key = key;
        this.tempo = tempo;
        this.lenMins = lenMins;
        this.lenSecs = lenSecs; // Constructor for the enums above
    }
}
```

```
for (AlkanMinorEtudes a : (AlkanMinorEtudes.values()) {
    System.out.println(a.replace('_', ' ') + " (" + a.key + " minor, " + a.tempo + ", " +
    a.lenMins + ":" + a.lenSecs + ")");
}
// COMME LE VENT (A minor, Prestissimamente, 4:35)
// EN RYTHME MOLOSSIQUE (D minor, Risoluto, 8:14)
// SCHERZO DIABOLICO (G minor, Prestissimo, 4:32)
// …
```

Applications of enums include days of the week, planets of the solar system, compass directions, etc.

Other Java Tools

The Java equivalent of Python's `datetime` is the `java.time` package, with some useful classes.

```
import java.time.*;
// The parent class to these classes is Temporal
LocalDate.now();        // 2022-06-29           -> supports operations down to year/mon/week/day
LocalTime.now();        // 20:13:46.342606500   -> supports operations up to sec/min/hour
LocalDateTime.now();    // 2022-06-29T20:13:46.342606500  -> supports both types of operations
OffsetDateTime.now();   // 2022-06-29T20:13:46.342606500-04:00
ZonedDateTime.now();    // 2022-06-29T20:13:46.342606500-04:00[America/New_York]
// toLocalDate(), toLocalTime(), … can convert upwards from more verbose to less verbose classes
Year.now();             // 2022
YearMonth.now();        // 2022-06
MonthDay.now();         // --06-29

// The following functions work for ZonedDateTime, but most should work for the other classes.
ZonedDateTime t = ZonedDateTime.of(2022, 6, 29, 20, 13, 46, 342606500, ZoneId.systemDefault());
// For OffsetDateTime, pass a ZoneOffset instead. You can do ZoneOffset.of("±HH:MM:SS")
// You can use ZonedDateTime.parse("2007-12-03T10:15:30+01:00[Europe/Paris]") and pass in a
// DateTimeFormatter object as an extra argument.
t.getDayOfMonth();  // 29
t.getDayOfWeek();   // WEDNESDAY, an enum of DayOfWeek
t.getDayOfYear();   // 180
t.getNano();        // 342606500
t.getSecond();      // 46
t.getMinute();      // 13
t.getHour();        // 20
t.getMonth();       // JUNE, an enum of Month
t.getMonthValue();  // 6
t.getYear();        // 2022
t.getOffset();      // -04:00 of class ZoneOffset
t.getZone();        // America/New_York, of class ZoneID
// time.plusNanos/Seconds/Minutes/Hours/Days/Weeks/Months/Years(long i) adds time
// time.minusNanos/Seconds/Minutes/Hours/Days/Weeks/Months/Years(long i) subtracts time

// Class TemporalAmount has children Duration (secs/nanosecs) and Period (years/months/days)
// Both have essentially the same methods, so I'll just focus on Duration.
Duration e1 = Duration.between(LocalTime.now(), LocalTime.now().plusSeconds(5));  // 5 secs
Duration e2 = Duration.ofSeconds(10); // 10 secs
// There's also ofDays(), ofHours(), ofMillis(), ofMinutes(), etc.
e1.compareTo(e2);   // -1 (this means e1 < e2)
e1.dividedBy(5);    // Duration object of 1 second
e1.multipliedBy(5); // Duration object of 25 seconds
e1.negated();       // Duration object of -5 second
e1.isNegative();    // false
e1.isZero();        // false
t.add(e1);          // ZonedDateTime object of 5 seconds after t; minus() works the same way.
// e1.plusNanos/Seconds/…() and e1.minusNanos/Seconds/…() works the same way
// e1.getNanos/Seconds() returns the duration of e1 in the specified unit
```

```
import java.time.temporal.*;
// The enum ChronoUnit can be inputted as arguments in certain functions.
// ChronoUnits has DECADES, YEARS, MONTHS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, etc.

e1 = Duration.of(5, ChronoUnit.MINUTES);
e1.get(ChronoUnit.SECONDS);               // 300. Note this only supports whole number returns.
e1.minus(3, ChronoUnit.SECONDS);          // Get a duration of 297 seconds
t.plus(1, ChronoUnit.CENTURIES);          // We're now in 2122
LocalTime.now().until(LocalTime.now().plusSeconds(1), ChronoUnit.MINUTES);       // 0 (rounded)
LocalDateTime.now().until(LocalDateTime.now().plusDays(1), ChronoUnit.SECONDS);  // 86400
```

One Java equivalent of Python's `input()` requires the `Scanner` class, in `java.util`.

```
import java.util.*;
Scanner scanner = new Scanner(System.in);
String input = Scanner.nextLine();   // Prompts the user for an input
// You can alternatively use the methods nextBoolean/Byte/Double/Float/Int/Long/Short().
// If the user enters an invalid result, InputMismatchException will be thrown
```

Alternatively, the scanner can also read .txt and .csv files.

```
// For .txt files
Scanner reader = new Scanner(new File("bee movie script.txt"));
while (reader.hasNextLine()) {
    String line = reader.nextLine();  // Files are processed line-by-line
    // Do something with line.
}
reader.close();

// For .csv files
Scanner reader = new Scanner(new File("top_ten_anime_betrayals.csv"));
reader.useDelimiter(",");   // CSV means comma-separated-value, so cells are separated by ","
while (reader.hasNext()) {
    String cell = reader.next();  // CSVs are processed cell-by-cell
    // Do something with cell. You can do nextInt() instead if you know the cell's an int.
    // If you don't want to process by cell, you can get rid of useDelimiter(",")
}
reader.close();
```

Another way to accomplish both tasks is with a `BufferedReader` class, in `java.io`.

```
import java.io.*;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String input = br.readLine();
```

And for reading txt/csv files…

```
BufferedReader br = new BufferedReader(new FileReader(new File("five or six stores.txt")));
while (String line = br.readLine() != null) { // Sets line to the next line, checks if not null
    // Do something with line
    // In a CSV, to split into cells, you might want to do String[] row = line.split(",");
}
br.close();
```

Both throw `IOExceptions` when there're errors in the file, plus `FileNotFoundExceptions`. Both are very similar overall; `Scanner` has some more functionality, but is slower than `BufferedReader`. `BufferedReader` can reader larger files too.