Basics of Java

Alexis Abbott Notes

Table of Contents

Basics

Components of a Class

Properties
Methods

These are considered members of the class.

Every Java program needs a main method.

Access Modifiers

Think of access modifiers as permission levels.

- public -
- protected - works like private, but enables access to subclasses in other packages.
- No modifier -
- private -

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	world
public	✔	✔	✔	✔	✔
protected	✔	✔	✔	✔	
no modifier	✔	✔	✔		
private	✔				

Errors

Compile Time

Compile time errors will happen during the compiling of an application right before it run when you start the application. Since Java is a strongly typed language it tries to prevent a lot of errors even before you get to the compiler, but they still happen. The stack trace will serve the error for debugging.

Runtime

Runtime errors are what happens when the application is executed. Java takes a lot of care to prevent errors before the program executes, but often these are logic errors or bugs. Error reporting helps developers and users alike to know what went wrong.

Data Types

- Primitives
- Objects

Primitive Types:

- boolean
- byte(8 bits)
- char - unicode(16 bits)
- short(16 bits)
- int(32 bits)
- long(64 bits) 9999999999L
- float(32 bits) 32.4f
- double(64 bits) more precise than float

Objects:

- Any time you call a constructor you're creating an object
- Class is the data type or interface

We can mix data types inside of expressions with the use of Type Casting. However, one needs to be aware of size limits and assign an appropriately sized variable.

This is fine

```
short a = 30;
double b = 9.5;

double c = a * b;
```

This is not

```
short a = 30;
double b = 9.5;

short c = a * b;
```

Doing it this way means you will lose data.

Variables

Expression - when you combine values and it evaluates to a value. The underlined below is an example.

```
int num1 = 50;
int num2 = 10;

int num3 = num1 + num2;
```

Literals - the value

```
String foo = new String("This is my string.");
```

Can convert to:

```
String foo = "This is my string.";
```

You can create a constant with the *Final* keyword.

Autoboxing

```
int x = 5; //primitive
Integer y = 5; //object
```

Each primitive type has the option to be converted to an object with their class equivalents. Despite the fact that both of the above variables are assigned to the number 5, one is a primitive and the other is an object. The reason to make a primitive an object is to gain access to the class methods (static utility methods).

Declaring a reference variable:

```
MyClass objRef = new MyClass();
```

Creating the object:

```
MyClass objRef = new MyClass();
```

Link the object to the reference:

```
MyClass objRef = new MyClass();
```

Calling the constructor:

```
MyClass objRef = new MyClass();
```

It’s the method that runs when you instantiate the object. It doesn’t always need to be implicitly declared.

Type Casting

Earlier we gave two examples of mixing data types in an expression:

This is fine

```
short a = 30;
double b = 9.5;

double c = a * b;
```

This is not

```
short a = 30;
double b = 9.5;

short c = a * b;
```

This is called Type Casting, where you imply conversion of one data type to another. You can do a widening type cast or a narrow type cast, but it’s important to keep in mind size constraints.

You can also cast manually:

```
double num = 10.2;
int x = (int)num;
```

Operand/Operator

Unary Operator

```
int result = +1;
```

The operator is only working on one operand. (result)

Binary Operators

```
int a = 5;
int b = 7;
int result = a * b;
```

The operator is working on two operands. (+ / *)

Ternary Operators

```
                        (true)         (false)
boolean expression ? expression1 : expression2
```

Generally set to a variable.

It’s helpful to make operands of the same type.

Comparison Operator ==

Works with primitives but not with objects, because the value of objects are references to the object memory/location.

Interning can cause unexpected results. If you have the same value in multiple string objects, the compiler interns it for you and uses the same memory location. It’s done to reduce the overall size of the program. On string literals, if you create objects with the constructor, it allocates the value to new memory.

Increment and Decrement

Incrementing uses two plus signs: x++
Decrementing uses two minus signs: x--

Prefix vs suffix
++x   x++

```
Int x = 5;
Int a = x++;
System.out.println(x);
System.out.println(a);
```

Expected output:

6
5

```
Int x = 5;
Int a = ++x;
System.out.println(x);
System.out.println(a);
```

Expected output:

```
6
6
```

Strings

Single quotes are reserved for characters and not interchangeable in Java.

String.format - good for templating

Taken from C/C++

%d - decimal	%s - string
%f - float	%t - date/time
%x - hexadecimal	%b - “true” if non-null, “false” if null
%c - character	...more

```
String output = String.format("%s = %d", "joe", 35);
```

To compare strings together, instead of using the comparison operator ==, use the equals() method instead.

LOOPS

Break keyword stops the loop whereas Continue skips that iteration.

Arrays

Syntax

```
datatype[] name = new datatype[10];

datatype[] name = {a, b, c, d, e};
```

Size of arrays are immutable.

2D Array

Column

Row	[0], [0]	[0], [1]	[0], [2]	[0], [3]	[0], [4]
	[1], [0]	[1], [1]	[1], [2]	[1], [3]	[1], [4]
	[2], [0]	[2], [1]	[2], [2]	[2], [3]	[2], [4]

datatype[ ][ ] name = new datatype [x][x];
The first index is the row and the second index is the column.

2D arrays can be various sizes and shouldn't be assumed to be square.

Column

Row	[0], [0]	[0], [1]	[0], [2]	[0], [3]	[0], [4]	[0], [5]
	[1], [0]	[1], [1]	[1], [2]	X	X	X
	[2], [0]	[2], [1]	[2], [2]	[2], [3]	[2], [4]	X
	[3], [0]	X	X	X	X	X

The empty cells with red x markers would not return values.

ArrayList

Syntax

```
List<datatype> name = new ArrayList<datatype>();
name.add(value1);
name.add(value2);
name.add(value3);
etc...
```

Create an ArrayList with an array

```
List<datatype> name = Arrays.asList(a, b, c, d, e, f);
```

Can loop to dynamically add items to the ArrayList.

ArrayLists don't work with polymorphism unless you use a wildcard. However, to keep from adding a wrong type to a list, using the wildcard means you can't add anything to the list. It becomes immutable.

Enum

An enum is a variable with a variety of possible values. Often we'll see them with switch statements. We use them when we know all possible values at compile time.

```
enum Day{
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}

public class Today{
  private Day day;

  public Today(Day whatDay){
    this.day = whatDay;
  }

  public static void main(String[] args){
    Today dayIs = new Today(Day.valueOf("SATURDAY"));
    dayIs.DayOfWeek();
  }

  public void DayOfWeek(){
    switch(day){
      case SUNDAY:
      case SATURDAY:
        System.out.println("Enjoy your weekend.");
        break;
      default:
        System.out.println("I bet you're working. Sorry.");
        break;
    }
  }
}
```

Every enum is implicitly **public static final** and since it's final, we can access it by its name.

Since enum extends the Enum class, there are a variety of helpful methods that packages with it.

Token

A file's bits are called tokens. They can be:

- Keywords
- Identifiers
- Constants
- Special Symbols
- Operators

Tuple

Data structures that allow you to group related info as a single thing. They are immutable.
["Notes", 0.3, "Alexis Abbott"]

Etc

To start a program from command line:

```
 $ java NameOfClassWMainMethod args
```

Java Packages that have x were extensions that got promoted to official library status. Ex:

javax.swing

Methods

Components

Argument: what you pass into a method call.
Parameter: definition of arguments should be passed in a method call.

Return: return keyword ends whatever method you're in.

Abstract

A method with the keyword abstract is declared without implementation and ends in a semicolon. If a method is declared abstract, the class must also be declared abstract.

```
abstract void myMethod(parameters);
```

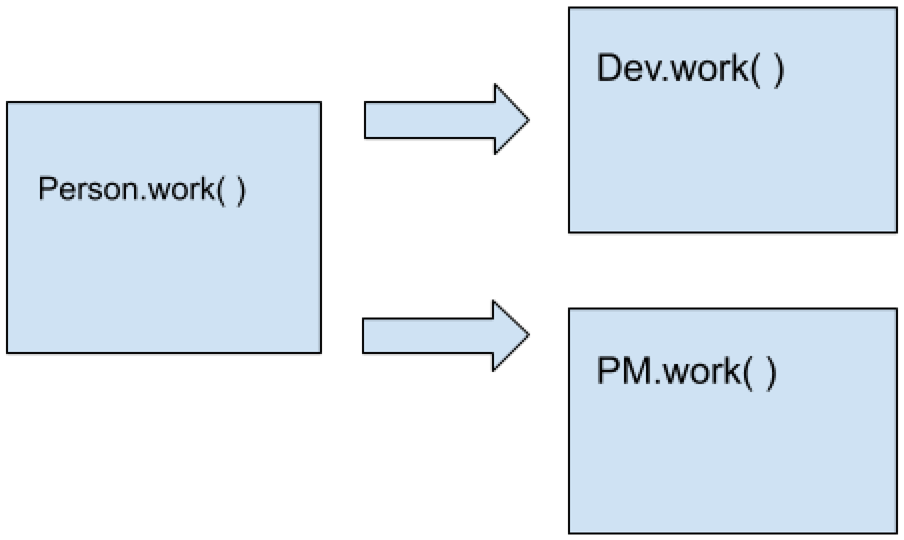
Overloaded Functions

Methods that can have the same name, but take in different parameters. Changing the parameters changes the method signature.

Changing the return type is not enough to change the method signature, the list of parameters also has to change. If the only change is return type then the method name will need to be changed. However, changing the order of the parameter list IS a legal example of changing method signatures for overloaded methods.

Override

Subclasses can override methods to change how they behave.



An example of this that happens a lot is overriding .toString( ) to output information about the object.

Methods that aren't static are considered virtual. This allows them to be overridden.

If a method has the abstract keyword it can have no body and MUST be overridden.

Encapsulation

By keeping classes self contained, we keep code streamlined, maintainable, easy to test, limits dependencies and therefore code breakage with changes. Narrowing access to getters and setters means internal logic can change without having resounding negative consequences downstream. Think of encapsulation as a container.

Scope

Think of scope as access to variable and object references and the timeframe in which they are available.

Closures

Because of the way the stack works in holding information until the top of the stack starts to "pop" off, access to variables further up the stack is viable.

```
public class SomeClass{
  public static String string = "This is a string.";

  public static void main(String[] args){
    System.out.println(string);
    changeString();
    System.out.println(string);
  }

  public static void changeString(){
    string = "Now a different string.";
  }
}
```

Expected output:
"This is a string."
"Now a different string."

Local variables being used in a function nested in scope are considered Final by the compiler.

```
public static void main(String[] args){
  int a = 10;
  int b = 20; //considered final

  doProcess(a, new AnonClass(){
    @Override
    public void process(int i){
      System.out.println(i + b); //cannot change this variable
    }
  });
}
```

Getter/Setter

Methods inside of an object class setup to interact with instance variables (fields). This prohibits access to them outside of the class and also normalizes how they're interacted with as they can only be accessed in particular ways through the getter and setter methods. Usually they are named getField and setField.

Getter

```
public datatype getSomeData() {
  return datatype;
}
```

Setter

```
public void setSomeData(param){
  //code
}
```

Static

Static methods can be accessed directly by any piece of code that has access to the class.

These are used on the Class and not the instance/object. They are in fact NOT part of a class instance and should only be used when the method applies to the Class rather than the instance.

Since they aren't part of the instance, this also means they don't have access to instance variables (fields) or instance methods.

Final

If a method should not be overridden, attach the final keyword in the declaration. It means subclasses have to use the inherited method.

```
public final datatype name( ){ //code }
```

The same can be done for classes, to make sure they can't be inherited (extended).

Try/catch

```
try{
  //some code
}catch(ExceptionName var){
  //do something with var for error reporting
}
```

When a method could be risky, it declares it will throw an exception.

```
public void riskyCode() throws ExceptionName{
  throw new ExceptionName("Useful info here.");
}
```

These exceptions inherit from the class Exception. If you throw an exception in your code, you must declare it using the *throws* keyword in the method declaration.

If you call the method that throws an exception, it has to go in a try/catch block or ducking by declaring the exception themselves to pass it up the stack.

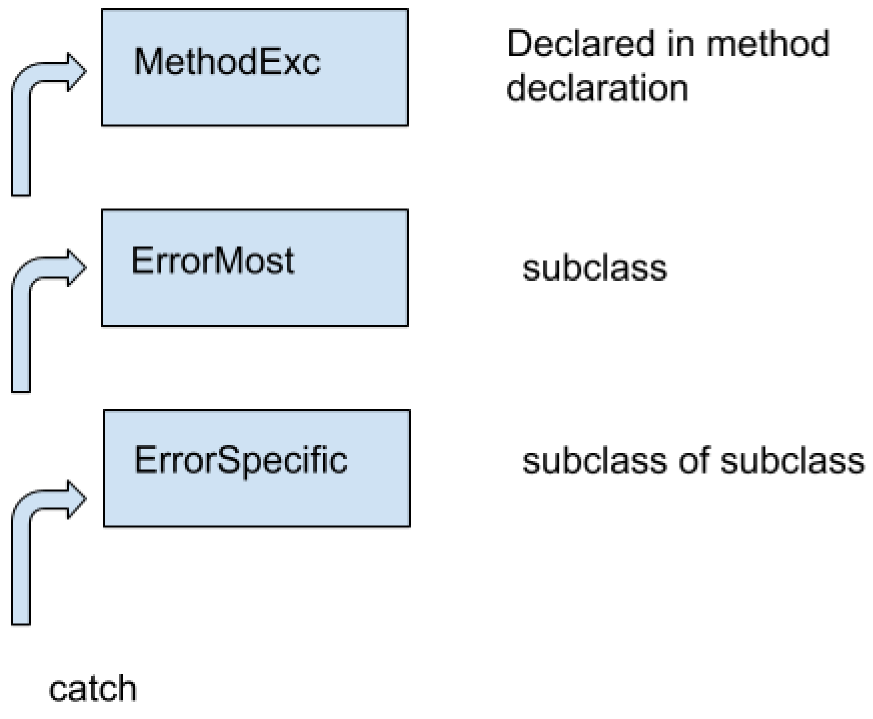
```
try{
  //some code
}catch(ExcError e){
  //handle the error
}finally{
  //this code runs regardless
}
```

Finally lets you put all of your important cleanup code in one place. If a method can throw more than one exception, you need to define a catch black for each one.

```
try{
  //some code
}catch(Err1 e1){
  //handle error 1
}catch(Err2 e2){
  //handle error 2
}
```

Exceptions are polymorphic. This means you can declare one exception in your method declaration and use subclasses to throw specific exceptions. You do not need to declare each one in the method declaration.

Catch blocks need to be ordered from smallest to biggest.



We must either handle or declare exceptions.

Exception Rules:

1. Can't have a catch or finally without a try.
2. Try must be followed by catch or finally.
3. Cannot put code between blocks.
4. A try with only a catch must duck the exception and declare it.

Lambda

Imperative - telling the program how to solve problems.
Declarative - telling the program what to solve.

Lambdas are:

- enable functional programming
- They're readable and concise
- Creates better APIs and libraries.
- parallel processing

The big difference between OOP and functional programming is OOP looks at code as things and nouns and functional programming looks at code as actions and verbs.

Functions can be assigned as values.

```
interfaceName functionAsVal = ( ) -> { //code }
```

If it can be contained on a single line, you can drop the curly braces.

```
interfaceName functionAsVal = ( ) -> //code
```

Lambda functions use the interface. Mostly just need to make sure the method signature matches. The interface name becomes the data type.

The reason Lambdas use the interface system was to make backwards compatibility possible.

Lambdas can only have one unimplemented method in the interface. These are called functional interfaces and need to be notated with:

```
@FunctionalInterface
```

What's nice about them only being able to have a single function is that you don't have to declare what method in your expression, making the code light. Generally you're passing behaviors to your functions so they should also be logic light.

Lambdas can be in other lambdas and also return lambdas.

Using Lambdas in looping and passing off the control to the JRE makes it easy for the process run in multiple threads. They can also be used to set up different operations in the stream.

Anonymous Inner Class

Lambdas are not anonymous inner classes. Lambdas do not change the state of *this*, whereas anonymous inner classes do.

Just like Lambdas, anonymous inner classes also can only have one method.

The syntax for an anonymous inner class is the same as invoking a constructor, but you also add a block of code.

```
public class MyApp() {
  public static void main(String[] args){
    SampleObj objName = new SampleObj(){
      @Override
      public void useOnce(){
        //simple task that doesn't need a whole new class
      }
    }
  }
}
```

Method Reference

A method reference can be done if your lambda is passing no arguments.

```
Thread t = new Thread(()-> printMesage());
t.start();
```

or

```
Thread t = new Thread(ClassName::printMessage());
t.start();
```

These snippets are both functionally the same.

Classes/Objects

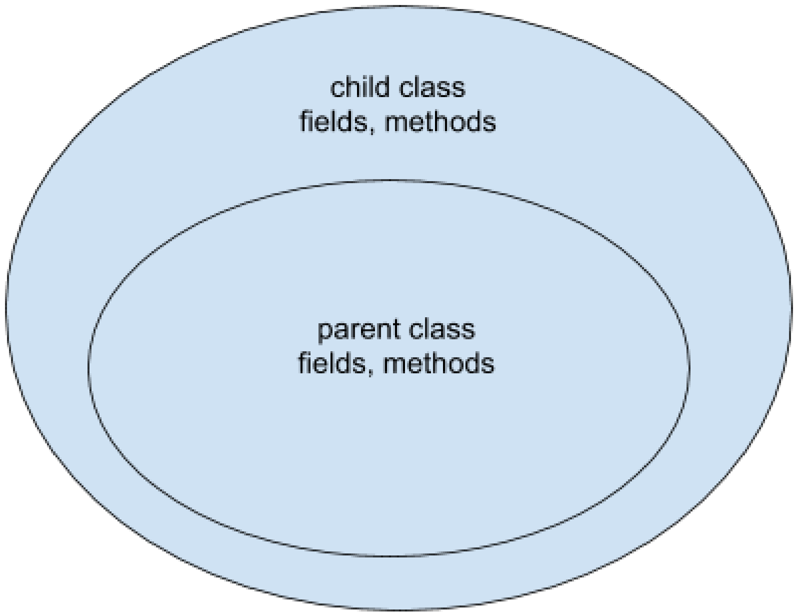
An object is an instance of a class. Create objects with the *new* keyword.

The structure for creating a new object:

```
type identifier = new type();
```

When this syntax is used, type() calls the Constructor of the class.

When an object is created, it becomes layered with instances of the inherited classes. This means we need space for both class instance variables. The child class wraps the parent class.



This means all constructors run when an object is created. Saying *new* is substantial as it starts the constructor chain reaction. Even abstract classes have constructors. This is called constructor chaining.

If you don't want an instance to be able to be called on a class, set the constructor to private. Math class is an example of this. You cannot make an object using the math class. Because the methods are static and public, they can be called directly.

```
int x = Math.round(42.2);
```

Often, a class with static methods is not meant to be instantiated. (Not always, the main method of your app is an example.) Usually the class is a utility with a set of behaviors.

What is a behavior? Algorithms in methods.

You should not call static methods from a variable reference.

Static Variable

A value shared by all instances of a class, instead of one value per instance. Static variables are initialized when a class is loaded, before an object is created, and before a static method runs. Static final variables are constants and can never change.

Math.PI

```
public static final datatype NAME = value;
```

This is the only way to create a constant and the naming convention is all uppercase.

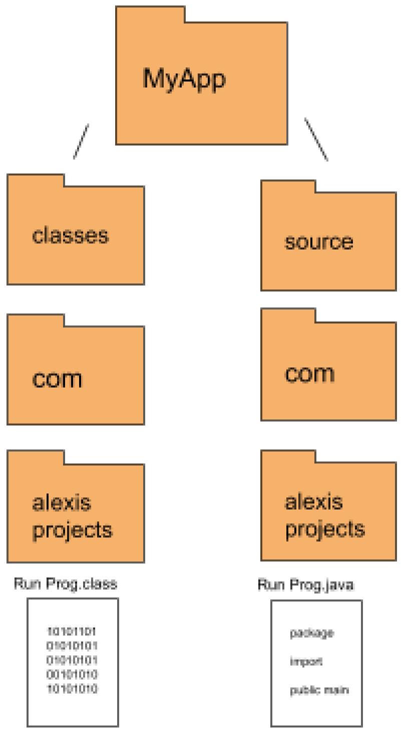
Package

This is a collection of classes in your project that give each other access to each other. The naming convention is the reverse of your domain name. Ex: com.alexisprojects.packageName

Package declarations must go above import statements.

If using the reverse domain method, you also have to match the directory structure. Example:

```
package com.alexisprojects
```



Import

Importing packages helps us from needing to fully qualify names. It also keeps our code readable.

```
import java.util.Scanner
```

This allows us to shorten to just Scanner vs java.util.Scanner

Fields

A variable in a class that isn't a method. They're instance properties that are usually manipulated through getter and setter methods. They're given default values even if you don't declare them. It's rare that they're set to anything other than private as other code should not be able to access them outside of the class.

Abstract

Putting abstract on a class prohibits instantiation and forces the use of subclasses. A conceptual way to look at this is that an "animal" does not exist, but specific examples of animal do: dog, cat, zabra, kangaroo, etc.

You can only extend one abstract class.

Interface

A way to achieve abstraction. It groups methods without bodies to contract a class that uses it into using the same methods.

Syntax

```
public interface Animal{
  public void talk();
  public int walk(int w);
  public void sleep();
}
```

In order for a class to use an interface, it needs to *implement* it.

```
public class Cat implements Animal{
  private int walk;
  private boolean atNight;

  public void talk(){
    System.out.println("Meow");
  };
  public int walk(int w){
    int walk = w++;
    return walk;
  };
  public void sleep(){
    atNight = false;
    return;
  };
}

public class Dog implements Animal{
  private int walk;
  private boolean atNight;

  public void talk(){
    System.out.println("Bark");
  };
  public int walk(int w){
    int walk = w++;
    return walk;
  };
  public void sleep(){
    atNight = true;
    return;
  };
}
```

The purpose is to avoid inheritance, it's a way to reduce maintenance especially when a new class can do the job better.

Interfaces don't have fields unless they're Static Final. Avoid doing this.

Functional Interface

A functional interface is used for functional programming and only has one method. For more information, look at the Lambda section.

Static

When declared, this class is directly accessible by any code.

Constructor

The constructor's purpose is to create an instance of a class. If you don't create a constructor method, the default will be used. If you create a custom constructor and still want the option of the default, just define it implicitly and leave the body empty. It needs to match the name of the class and not have a return type. If you create an instance using the new keyword and don't pass arguments, the default constructor will be used.

Whenever you create an instance of a class, you're actually calling the Constructor.

A method in a class, named the same as the class, without a return type, is the constructor. They *cannot* have a return type. Constructors are not inherited and you can overload constructors.

POJO

Plain Old Java Object

Used for increased readability and re-usability, the Java specifications are:
1. DON'T extend prespecified classes
2. DON'T implement prespecified interfaces
3. DON'T contain prespecified annotations.
4. DO be public.
5. DO make fields private.
6. DO have a no argument constructor (default)
7. DO have getters and setters

Beans have an extra requirement:
- Implement Serializable or Externalizable interface.

Java Beans

Classes that encapsulate many objects into a single object. They're POJOs that must be serializable. It's just a standard.

Reference Variables

Whenever a variable is set to an object, it becomes known as a reference variable. It also behaves very differently from other variables.

Passing Values vs Reference

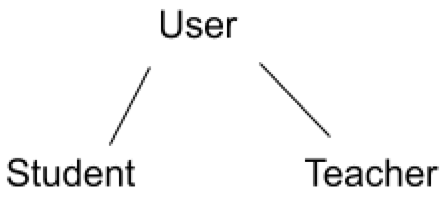
When sending a variable to a method you're passing the value, not the actual variable. If you make changes to that value inside of the method, it DOES NOT make changes to the variable outside of the method.

However when passing an object reference, what is being passed is the memory location and so changing the object inside of a method will impact its uses outside of the method. It's important to keep this in mind when passing arguments.

If the original object shouldn't be altered inside of a method, it should instead be cast to a new instance inside of the method.

Polymorphism

The ability for something to be multiple things.



Since Student and Teacher are subclasses of User, we can group instances of each. It also makes things like this possible:

```
User kid = new Student();

Student kid = new Student();
```

Example:

```
Student student = new Student();
Teacher teacher = new Teacher();

List<User> users = new ArrayList<User>();
users.add(student);
users.add(teacher);
```

Since student and teacher are both of type User, they can be collected in a User ArrayList.

super

If you need things from the parent class, use the super keyword. You can access instance variables, methods, and even the parent constructor.

Instance Variable

```
class Animal{
  private String name = “Animal”;
}

class Dog extends Animal{
  private String name = “Fido”;

  public void nameTest(){
    System.out.println(name); //Fido
    System.out.println(Super.name); //Animal
  }
}

class MyAnimalApp{
  public static void main(String args[]){
    Dog d = new Dog();
    d.nameTest();
  }
}
```

Class Method

```
class Animal{
  public void talk(){
    System.out.println(“I’m such an animal.”);
  }
}

class Dog extends Animal{
  public void talk(){
    System.out.println(“Bark.”);
  }

  public changeBack(){
    Super.talk();
  }
}

class MyAnimalApp{
  public static void main(String args[]){
    Dog d = new Dog();
    d.changeBack();
  }
}
```

Constructor

```
class Animal{
  private int height;
  private int weight;

  public Animal(int h, int w){
    height = h;
    weight = w;
  }

  public String toString(){
    return "Animal: Height: " + height + " Weight: " + weight;
  }
}

class Dog extends Animal{
  private String collarColor;
  public Dog(int h, int w, color){
    super(h, w);
    this.collarColor = color;
  }

  public String features(){
    return "Dog: Collar: " + this.collarColor + " " + super.toString();
  }
}

class MyAnimalApp{
  public static void main(String args[]){
    Dog d = new Dog(7, 30, "pink");
    System.out.println(d.features());
  }
}
```

If you can reuse logic from the super class, it makes sense to pass what you need to the super constructor to keep from repeating code.

The compiler adds super(); if you don't call it implicitly. If you need to use an overloaded constructor from the super class, you need to call it implicitly as the compiler with only ever grab the no arguments constructor. It has to be the first thing called in your child constructor if you do.

Note: this(); can also be called in a constructor and also has to be first if used. Therefore you can have this() or super(), but never both.

Object Graph

When an object is serialized, if it has any references to other objects, those will also be serialized. The whole mapping is considered the Object Graph.

Constructor Chaining

When creating a new instance of a class, the constructor is invoked. If that class has any parent classes, those constructors are also called, and if they have parents, those constructors are called, and so on.

Serialized

A keyword to add to a class in order to make the object "savable". When an object is "flattened" to be passed over TCP/IP or to save as a text file locally. More information in the I/O section.

Singleton

You can create "Singletons", restricting a class to making only one instance. It's an OOP pattern by using a static getter method and a private constructor.

Generics

Anything in Java that makes use of angle brackets are called generics. <T> Type variable.

Virtually all of the code using generics will be part of Collections. It's a way to write type safe code. This means problems are found at the compiler rather than runtime.

It's a class that can take a type (not primitives) and define that through the class.

```
public <T extends Animal> void useObj(ArrayList<T> list);
```

By adding a type parameter before the return type, it says T can be any type of Animal. It's not the same as this though:

```
public void useObj(ArrayList<Animal> list);
```

Both are valid, but they're different. The second example only allows for Animal - no subclasses.

In generics, "extends" means "extends or implements". The reason for using extends in the generic declaration to represent "is-a" is to avoid creating a new keyword. This could break backwards compatibility.

List: when SEQUENCE matters. Collections that know about index position.
Set: when UNIQUENESS matters. Collections that do not allow duplicates.
Map: when you need KEY-VALUE pairs. You can't have duplicate keys.

Hashcode

<u>Reference Equality vs Object Equality</u>

- Reference Equality
    - Two references, one object on the heap.
- Object Equality
    - If you decide two objects are the same because they both share the same title, as an example, you must override both hashCode() and equals() methods. (Inherited from class Object.)
    - Two references, two objects on the heap, but the objects are considered meaningfully equivalent.

Most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode. The default behavior is to give each object in hashCode() a unique hashcode. This means you need to override so equivalent objects will return the same hashcode.

You must also make sure equals() will return true on either object.

HashSet

When you put an object into HashSet, it uses the object's hashcode value to determine where to put the object in the set. It checks the hashcode against all objects and assumes this is a new object.

If HashSet finds a matching hashcode for two objects - one you're inserting and one already in the set - the HashSet will then call one of the object's equals() methods to see if these hashcode-matched objects really ARE equal.

Object Law

for

hashCode() and equals()

- If two objects are equal, they MUST have matching hashcodes.
- If two objects are equal, calling equals() on either object must return true.
  obj1 ⥃ obj2
- If 2 objects have the same hashcode value, they are NOT required to be equal, but if they are equal, they must have the same hashcode value.
- If you override equals(), you MUST override hashCode().
- The default behavior of hashCode() is to generate unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects can ever be considered equal.
- The default behavior of equals() is to do an == comparison. In other words, to test whether two references refer to a single object on the heap. If you don't override equals, no two objects can ever be equal since references to two objects will always contain a different bit pattern.
  a.equals(b) must also mean
  a.hashCode() == b.hashCode()
  But, a.hashCode() == b.hashCode() does not have to mean a.equals(b).

TreeSet

TreeSet is like HashSet, but things stay sorted in TreeSet. It's a bit more of a performance hog however.

HashMap

```
HashMap<String, Integer> scores = new HashMap<String, Integer>();
scores.put("Jake",54);
scores.put("Todd",72);
```

Optional

A class that allows for optional values instead of null references. It supplies several methods to check our values against so we can instead assign something meaningful.

Garbage Collection

Life Cycle of Object

When JVM starts, it gets a chunk of memory from the underlying OS. How much memory depends on your JVM and which platform you're running. Tweaking this often doesn't matter.

The method on top of the stack is the current executing method. When a method is finished it gets "popped" off the stack.

References of objects are stored in the Stack, not the object itself.

Knowing the fundamentals of Stack and Heap is crucial to variable scope, object creation issues, memory management, threads, and exception handling.

An object remains "alive" as long as a reference variable exists in the stack. Once the reference variables are gone, the Garbage Collector can now get rid of the object, freeing memory. This means abandoning objects to the Garbage Collector can be an important part of optimization and prevent out-of-memory death.

Object Killers

1. Reference goes out of scope permanently.
2. Assigning that reference to another object.
3. Reference is set to null.

Heap

Where objects live.

Stack

Where the method invocations and local variables live.
The stack is very much like a stack of boxes. You must deal with the one on top and go down sequentially.

I/O

Need two streams, one to represent the connection and another to call methods on. To use them together, you can chain streams.

Serialization

When an object is serialized, it's "flattened". It's then written to bytes and saved to a file. Serialization saves the entire object graph. When an object is saved, if it has instance variables that are reference variables, those objects are also serialized.

To make a class serializable, it must implement the Serializable Interface. This will apply to all subclasses even if they don't implicitly declare. It's in the java.io.* package, so it will need to be imported.

I/O operations can throw exceptions. Serialization is an all or nothing, so if an object in the graph can't be serialized, it fails.

Marking an instance variable (reference variable) as transient will skip it if it can't or shouldn't be saved. It will be brought back as null.

Sometimes an instance can't be saved in any meaningful way and has to be recreated from scratch. If the class isn't final, you can serialize a subclass that extends the class you need. What happens though is the superclass creates a new instance from scratch.

When bringing back an instance variable that is transient, there are two options:
1. Reinitialize the null instance variable to some default state.
2. If the variable matters, save the key values and recreate the object to be identical to the original.

When an object is deserialized, the class is loaded, but the constructor does not run. If there's a non-serializable class in the inheritance tree, the constructor for that will run.

For objects serialized to ship over a network, it can be stamped with a URL for where the class can be found. It's used in Java's Remote Method Invocation (RMI). You could send it as part of a method and if the JVM doesn't have the class, it will use the URL to fetch the class to load it automatically.

Don't make serializable objects dependent on dynamically changing static variables. It might not be the same when the object comes back.

Changes That Hurt Deserialization

- Deleting an instance variable.
- Changing declared type of instance variable.
- Changing non-transient to transient.
- Moving a class in the hierarchy tree.
- Changing a class in the tree from Serializable to not Serializable.
- Changing instance variable to static.

Changes That Don't Impact Deserialization

- Adding new instance variables. Will be given defaults when deserialized.
- Adding classes to the tree.
- Removing classes from the tree.
- Changing access levels of instance variables.
- Changing an instance variable from transient to non-transient.

Each time an object is serialized, it's given an ID called the serialVersionUID. It's computed based on information about the class structure. If the class has changed since the serialization happened, the serialVersionUID won't match and deserialization failes.

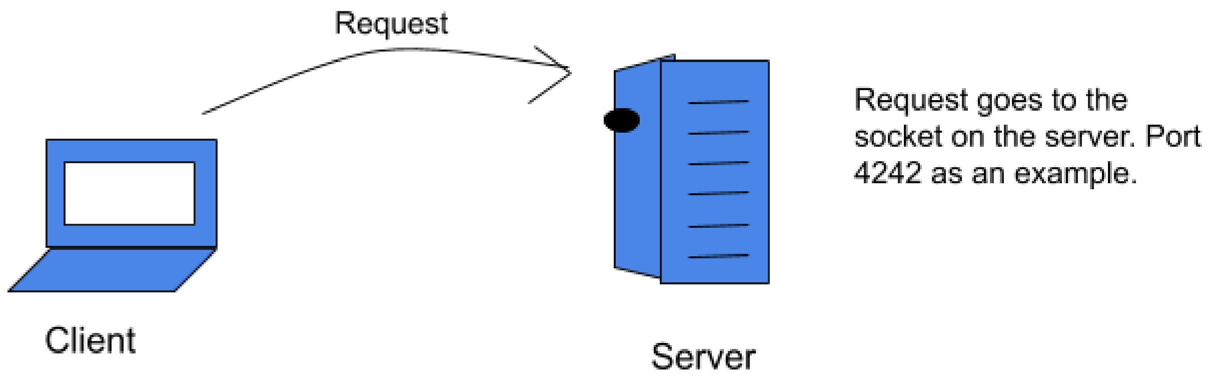
You can get around this by defining the serialVersionUID in your class. It however means you must be careful with what changes you make to the class. You can check for the serialVersionUID via command line. Just copy/paste it into your class.

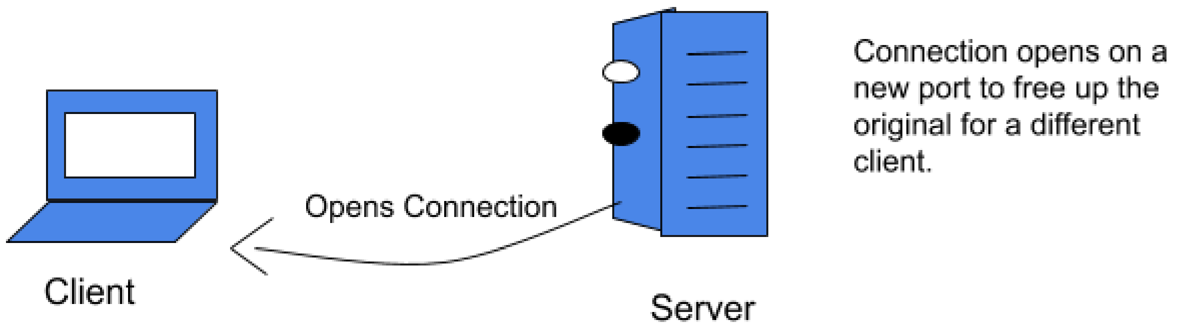
Sockets

An object that represents a network connection between two machines.

```
Socket con = new Socket("IP#", 8080);  //8080 is the port number
```

You can't have more than one application on a port. To connect over a socket, you use streams.


Request
Request goes to the socket on the server. Port 4242 as an example.
Client
Server


Connection opens on a new port to free up the original for a different client.
Opens Connection
Client
Server

Port Ranges

When picking a port number for your application, choose in the range of 1024 - 65535. 0 - 1023 are reserved for well known services.

RMI

Remote Method Invocation

Creating a helper object acts as a proxy with all the low level details of sockets and streams. The big difference between a local method call and a remote method call is I/O over a network. It throws exceptions.

The client helper is called the RMI Stub and the server helper is an RMI Skeleton.

Step 1: Make a Remote Interface. This will define what methods can be called remotely.
Step 2: Make a Remote Implementation. The class that does the real work. It's the object the client wants to call methods on.
Step 3: Generate the stubs and skeletons using rmic. Handled automatically by the tool in the JDK.
Step 4: Start the RMI Registry (rmiregistry). Where the user goes to get the proxy.

Step 5: Start the Remote Service. Service implementation class instantiates an instance of the service and registers it with the RMI Registry. Registering it makes it available for clients.

Interfaces can't implement other interfaces, but can extend them. Methods must throw an exception:

```
public interface AppRemote extends Remote{
  public String sayHello()throws RemoteException;
}
```

Be sure return values and arguments are primitives or serializable. The object has to be packaged and shipped across the internet and that happens with serialization.

Implement the remote interface:

```
public class RemoteImpl extends UnicastRemoteObj implements AppRemote{
  //code
}
```

UnicastRemoteObject is part of java.rmi.server package. It can do most of the heavy lifting for you.

Write a no-arg constructor that declares a remote exception. Register the service with the RMI Registry.

EJB (Enterprise Java Bean)
A Java class with one or more annotations from the EJB spec which grants the class special powers and when running inside the EJB container.

Jini is also possibly something to study if needed in the realm of RMI. It's a self-healing network that uses service leases to automatically remove services it can no longer access.

Saving to File

You use FileWriter instead of FileOutputStream and you don't need to chain it to anything.

To write a serialized object:

```
objectOutputStream.writeObject(someObject);
```

```
fileWriter.write("A string");
```

Buffers

Buffers give a temporary holding place until it's full.

In the case of FileWriter, doing things one at a time creates a lot of overhead. Using a buffer will hold things until it's full and then write to disk. You can empty the buffer before it's full using .flush()

Threads

Java is multi-threaded, meaning tasks can be run concurrently. It takes advantage of the fact that most computers now have several processors.

A thread can have a few states:

- runnable
- running
- temporarily not runnable (blocked)

There are a variety of reasons a thread can become blocked:

- Executing code to read from a socket input stream, but no data to read.
- Executing code might tell it to sleep.
- Tried to call a method on an object already in use.
- etc.

A runnable is to a thread what a job is to a worker.

Once the thread has completed its task, it's dead. It can not be called again. It may still be in the heap, but it's just an object ready for Garbage Collection!

Schedule

The JVM decides the thread schedule. You can influence it, but it is largely out of your hands. You cannot predict the JVM schedule and this needs to be accounted for in your program. Schedulers are even implemented differently in different JVMs. Running the same program on the same computer can give different results.

!NOOB MISTAKE!
Only testing a program on one machine, assuming the thread scheduler will always work that way.

The concept of Write-Once-Run-Anywhere means the program must work no matter how the thread scheduler behaves.

The secret here is sleep(). Even doing it for a few milliseconds forces the currently running thread to leave the running state. The only guarantee - sleep means the thread can't become active until the sleep time is over. sleep() throws an exception, so it must be wrapped in a try/catch block.

The thread, when it’s done sleeping, will become the active thread -> but it has to wait its turn. Don’t program thinking the sleep duration is accurate in regard to when it will complete its task.

Threads can lead to concurrency issues.

Synchronize

Using the synchronized keyword to modify a method means only one thread can access it at a time. This assures a thread can fully finish a method, even if it goes to sleep in the method.

Object Lock

Handled by the JVM, there’s no API for accessing object locks. When a thread hits a synchronized method, it recognizes it needs a key for the object before it can enter the method. It looks for the key and if it’s available, the thread grabs the key and enters the method.

From that point on, the thread retains control of the key until it completes the synchronized method. While a thread is holding the key, no other threads can enter any of that object’s synchronized methods because they can’t access the key.

If you have a method updating an increment, you can have multiple threads fighting over the value of your incrementer, causing havoc. A way to prevent this from happening is to make the process atomic (a whole thing) by synchronizing the method.

This however comes with overhead. If threads get bottled up at a synchronized method, your application can take a performance hit. You can synchronize at granular levels to avoid shutting down an entire method.

When working with multiple threads, it’s possible for them to get into a stale-mate with object keys resulting in deadlock. It’s possible to have your entire app stall while threads forever wait on each other. There are architecture design books on how to avoid deadlock as Java has no way of detecting this.

You can also synchronize a static method, a lock will be created for the class as a while instead of a single instance (object).

There are also thread priorities which offer no guarantee. Use priorities for enhancing performance only - not for program correctness.

MVC

Completely separates calculations (logic) and interface (view) from each other.

Model

Data and methods used to work with the instance. (Entity Class) This is also known as the persistence layer as it persists the state of the program.

Controller

Coordinates actions between the View and Model

View

The Interface