Lecture 1 - 6 Notes

Contents

- Mother of All Classes
- Classes within Classes
- Variable Capture
- Other Things to Take Note

6 - Inheritance and Polymorphism

- Interface
- Superclass and Subclass
- Quote
- Multiple Inheritance of Classes?
- Simplified Diagram of Java Memory Model with Superclasses
- Liskov Substitution Principle (covered in SOLID)
- Instantiation, Early Binding and Late Binding
- Inheritance vs Composition
- Abstract Classes
- Polymorphism

7 - Types and Subtypes

- Symbolic Execution
- All About Subtypes
- Primitive Types and Conversions
- Wrapper Class Caching

8 - Generics and Wildcard

- What are Generics?
- Bounded and Unbounded Generics
- How does Generics ensure Type Safety?
- What are Wildcards?
- Bounded Wildcards
- PECS
- Variance of Generics and Wildcards
- Type Erasure

9 - Exception Handling

- Keywords for Manual Exception Handling
- How does the Java Virtual Machine handle Exceptions?
- Catching Exceptions

10 - Java API and Packages

- Collections
- Map
- Summary of Various Data Structures

- **Equality Check with HashMap**
- **Scanner**
- **Packages**
- **Random**

11 - Enum

- **What is Enum?**
- **How does Enum ensure Type Safety?**
- **Values(), ordinal() and valueOf() methods**

Chapter 1: Recap on Basics of Programming

1.1 What is Programming?
Writing code that operates on some data. The code is a process that operates on data, which is a representation.

1.2 What is Data?
Data is a way of representing some meaning. For example, both characters and bytes (numbers), which are different meanings, can be represented as binary.

1.3 Natural Changes in Data
1.3.1 Change in Meaning without Change in Representation
0100 0110 => 70
0100 0110 => 'F'

1.3.2 Change in Representation with Minimal Changes in Meaning
70 as a byte => 0100 0110
70 as an integer => 0000 0000 0000 0000 0000 0000 0100 0110
70.0 as a float => 0100 0010 1000 1100 0000 0000 0000 0000

1.3.3 Changes in both Representation and Meaning
70.5 as a float => 0100 0010 1000 1101 0000 0000 0000 0000
70 as an integer => 0000 0000 0000 0000 0000 0000 0100 0110

1.4 What are the different styles of programming?
1.4.1 Imperative
"Let me tell you what to do."

- Process-based
- Involves:
    - Sequences: Do this then that
    - Selection: If this then that, else this
    - Repetition: While this do this

1.4.2 Object-Oriented
"Let me tell you what this object can do to that object"

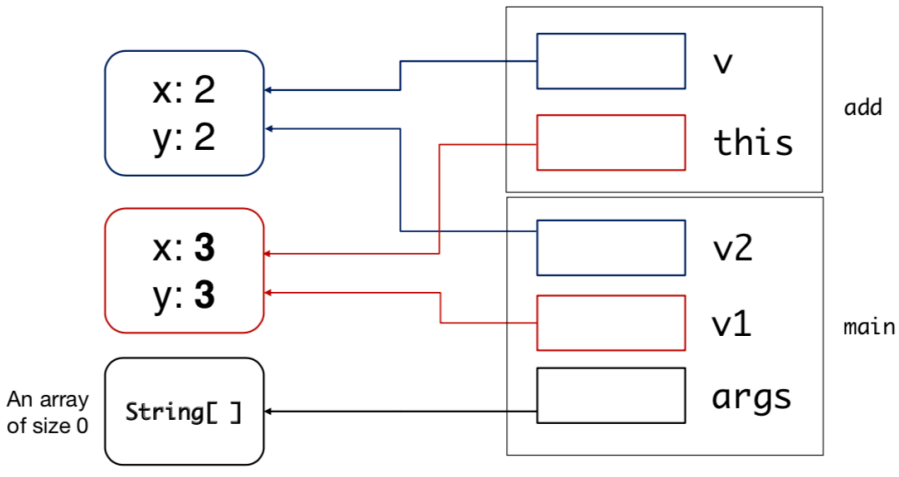
Uses both processes and data/representations

1.4.3 Functional
“Let me describe using this mathematical notation"
Process-based

1.4.4 Declarative
“I don’t care how you do it, but let me tell you what I want”
Data/representation-based

1.5 Stack and Heap

```
class Vector2D {
    private double x;
    private double y;
    Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }
    void add(Vector2D v) {
        this.x += v.x;
        this.y += v.y;
    }
}
class Main {
    public static void main(String[] args) {
        Vector2D v1 = new Vector2D(1, 1);
        Vector2D v2 = new Vector2D(2, 2);
        v1.add(v2);
    }
}
```



Chapter 2: Abstraction and Encapsulation

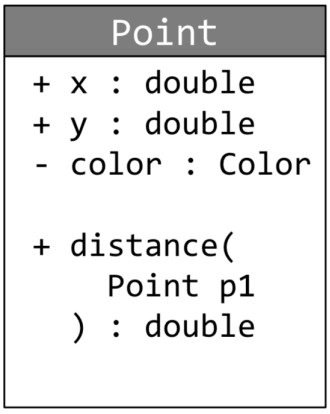
2.1 Functional Abstraction

The process of naming a composition of steps, which helps to hide control structures.

2.2 Data Abstraction
The process of separating the abstract properties of a data type and the concrete details of its implementation.

2.3 Encapsulation
The process of restricting direct access to some of an object's components, as well as the bundling of the object's data with the object's methods (or other functions) operating on that data.

2.4 Class Diagram (Also called Unified Modelling Language)



+ Public
- Private
Protected
~ Package-private/Default
Shaded Arrows for Composition
White Arrows for Inheritance

2.5 Functions to Write for HashKeys
If a class needs to be able to function as a HashKey for a HashMap, it needs the following two methods overridden:

- public boolean equals(Object o){}
- public int hashCode(){}
    - Because if x.equals(y), then x.hashCode() == y.hashCode()

Chapter 3: Good Programming Practice

3.1 Naming Conventions

- Class name: UpperCamelCase
- Method name: lowerCamelCase
- Field name: lowerCamelCase
- Constant name: ALL_CAPS_SNAKE_CASE

3.2 Bracketing and “Paragraphing”

Egyptian Bracket

```
if (x==0){
    x++;
}
```

3.3 Indentation

2 Spaces

3.4 Line Limit

- 80-100 character for each line
- If the line is longer than that, break at a suitable/logical point and indent at least 4 spaces

3.5 Coupling and Dependencies

3.5.1 Coupling

Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

Low/weak/loose coupling is often seen as a good thing, as tightly coupled systems tend to exhibit the following developmental characteristics, which are often seen as disadvantages:

- A change in one module usually forces a ripple effect of changes in other modules.
- Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
- A particular module might be harder to reuse and/or test because dependent modules must be included.

3.5.2 Dependency

Dependency is the level of reliance on other modules to work before the module in question will work. There are two main types of dependencies:

- Hard Dependencies
    - You need other classes to have the correct fields before the current class can work
    - Extension is also a hard dependency
- Soft Dependencies
    - You depend on other classes to have correct methods

Cyclic dependency occurs when two classes both need each other to work before they can function properly. This is often not a good programming practice, as it is difficult to identify which part of the code is not working.

3.6 Hierarchy
The advised way to structure one's code is based on a hierarchy, where there is a base level of dependencies that function on their own, and subsequent levels above that depend on the level below.

This allows us to test each part individually once its dependencies have been tested. You can also code from bottom-up, and everything is self-contained as much as possible i.e. everything makes sense on its own. As the bottom dependencies are independent from each other, teams can also work together with greater ease, so long the part they code achieves its function.

3.7 SOLID Principles
3.7.1 Single Responsibilities Principle
A class should have only one reason to change.
A class / module should do only one thing and do it well.

3.7.2 Open-Closed Principle
Software entities (classes, modules, methods, etc) should be open for extension, but closed for modification.
Open

- Available for extension
- Add new fields to data structures
- Add new elements to set of methods it performs

Closed

- Available for use by other modules
- Assumes has well-defined and stable description
- Try not to modify what is already done except for bug-fixing.
- Perhaps write a new function that uses an existing one.

Basically try not to code classes or methods that depends on "knowing it all" to work. Instead, try to construct templates through interfaces, abstraction or polymorphism/extension. This prevents the existence of "God Classes", or classes that need to know everything before it can work e.g. if-else with all possibilities crammed in one method.

3.7.3 Liskov Substitution Principle
Let ϕ x be a property provable about object x of type T. Then ϕ y should be true for object y of type S where S is a subtype of T. If S is a subclass of T, then object of type T can be replaced by an object of type S without changing the desirable property of the program.

Answer format:
Yes, it violates LSP. The subclass changes the behaviour of the superclass, so the property that (enter desirable property here) no longer holds for the subclass. Places in a

program where the superclass is used cannot be simply replaced by the subclass.

3.7.4 Interface Segregation Principle
No client should be forced to depend on methods it does not use.
Keep interfaces minimal

- do not force classes to implement methods they can't
- do not force clients to know of methods in classes they don't need to

Bad Example:

```
class Shape {
    public void print() { .. }
    public void draw() { .. }
}
class Drawer {
    private Shape[] shapes;
    public void drawAll() {
        for(Shape shape : shapes) {
            shape.print(); shape.draw();
        }
    }
}
```

Fixed Example:

```
class Shape implements Drawable, Printable {
    public void print() { .. }
    public void draw() { .. }
}
class Drawer {
    private Drawable[] drawables;
    public void drawAll() {
        for (Drawable drawable : drawables) {
            drawables.draw();
        }
    }
}
```

3.7.5 Dependency Inversion Principle
High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details.
Details should depend on abstractions.
Programming to an interface

- Not to an implementation
- Benefits
    - Maintainability
    - Extensibility

- Testability

3.8 Assertions

Assertions are assumptions about the behaviour. When the assertion/assumption is not met, the program terminates to stop it early, instead of letting the erroneous behaviour continue.

An example would be

```
public static double distance(Point x, Point y) {
    double distX = ..;
    double distY = ..;
    double dist = ..;
    assert dist >= 0;
    return dist;
}
```

If dist is <0 at the point of assertion, the program terminates. The cost of assertion is the cost of computing the values being asserted.

To enable assertions when running, the following command is required:

```
java -ea Main

// Sample error message:
Exception in thread “main” java.lang.AssertionError
        at AssertionTest.distance(AssertionTest.java:7)
        at AssertionTest.main(AssertionTest.java:3)
```

However, it may be difficult to decipher the error message, since it’s just an exception. As such, you can write your own error message:

```
assert booleanExpression : message

public static double distance(double x, double y) {
    double dist = x - y;
    assert dist >= 0 : "x: " + x + "; y: " + y;
    return dist;
}
```

Chapter 4: All about Functions

4.1 Method/Function Signature

A function has its function signature:

- Function name
- Parameter types
- Number of parameters

- NOT RETURN TYPE

4.1.1 Why is it safe to ignore the return type when considering method signature?
Java supports covariant return types for overridden methods. This means an overridden method may have a more specific return type. That is, as long as the new return type is assignable to the return type of the method you are overriding, it's allowed. The relationship between the two covariant return types is usually one which allows substitution of the one type with the other, following the Liskov substitution principle.

This is the exam answer which does not make sense: Existing code that has been written to invoke the superclass' method would still work if the code invokes the subclass' method instead after the subclass inherits from the superclass.

4.2 Method Overloading?
It's when a two functions have the same name but have:

- One or more different parameter types
- A different number of parameters

4.3 So which method will we call?
It calls the exact method e.g. if the input is 1, it'll call the method with an integer as a parameter. The reverse occurs if there's no alternative, e.g. (1) and there's only one method with double as a parameter available. Type promotion occurs, which is when a data type of smaller size is promoted to a data type of a bigger size e.g. byte to short, int to double.

If there is no clear result available e.g. 2 functions have equal suitability, then it will throw an error.

```
|  Error:
|  reference to f is ambiguous
|    both method f(int,double) in Try and method f(double,int) in Try match
|  a.f(1,1)
|  ^-^
```

4.4 Function Evaluation
In Java, all values of arguments are copied into parameters.

4.5 Method Overriding, Static Binding and Dynamic Binding
4.5.1 Method Overriding
Method Overriding occurs when a subclass overrides a superclass' method with the exact same method signature. For example,

```
class Human{
    //Overridden Method
    public void walk()
    {
        System.out.println("Human walks");
    }
```

```
}
class Demo extends Human{
    //Overriding Method
    @Override
    public void walk(){
        System.out.println("Boy walks");
    }
}
```

But how do we know what method is called? Let us see.

4.5.2 References

```
class Human{
....
}
class Boy extends Human{
    public static void main( String args[]) {
        //This statement simply creates an object of class Boy and assigns
a reference of Boy to it*/
        Boy obj1 = new Boy();

        // Since Boy extends Human class. The object creation can be done
in this way. Parent class reference can have child class reference assigned
to it
        Human obj2 = new Boy();
    }
}
```

4.5.3 Static Binding or Early Binding

So how do we know which method is called? The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is compile time. The reason is that the these method cannot be overridden and the type of the class is determined at the compile time. Let's see an example to understand this.

Here we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overridden so even though an object of Boy class is used while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there.

```
class Human{
    public static void walk()
    {
        System.out.println("Human walks");
    }
}
class Boy extends Human{
    public static void walk(){
```

```
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        // Reference is of Human type and object is Boy type
        Human obj = new Boy();
        // Reference is of Human type and object is of Human type.
        Human obj2 = new Human();
        obj.walk(); // Human walks
        obj2.walk(); // Human walks
    }
}
```

4.5.4 Dynamic Binding or Late Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed. The actual type of object is determined at the runtime so this is known as dynamic binding.

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are not static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens. Lets see an example to understand this:

```
class Human{
    //Overridden Method
    public void walk()
    {
        System.out.println("Human walks");
    }
}
class Boy extends Human{
    //Overriding Method
    public void walk(){
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        // Reference is of Human type and object is Boy type
        Human obj = new Boy();
        // Reference is of Human type and object is of Human type.
        Human obj2 = new Human();
        obj.walk(); // Boy walks
        obj2.walk(); // Human walks
    }
}
```

Chapter 5: Classes and Objects

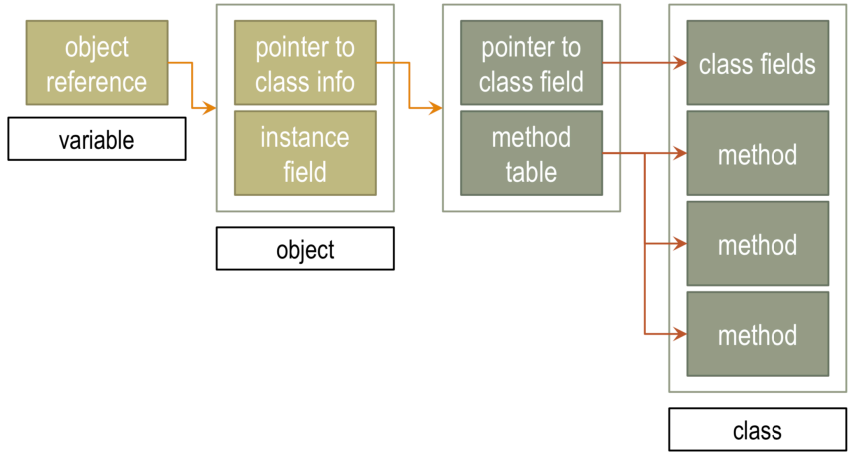
5.1 Anatomy of a Class

Fields
Constructors
Methods

5.2 Class vs Object
A class is a template for object creation. An object is an instance of a class.

5.3 Static Fields vs Instance Fields
A static field is a class-level field, or a field that belongs to the class rather than a specific instance. It can be accessed both through Class.field and Instance.field. However, instance fields cannot be accessed through the class, as the instance fields is unique for each object instance.

5.4 Simplified Diagram of the Java Memory Model



5.5 Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Table from: https://www.javatpoint.com/access-modifiers

5.6 Final Keyword
In the Java programming language, the final keyword is used in several contexts to define an entity that can only be assigned once.

- Final Class: Cannot be subclassed
- Final Method: Cannot be overridden or hidden by subclasses
- Final Variable: A final variable can only be initialized once, either via an initializer or an assignment statement. It does not need to be initialized at the point of declaration: this is called a "blank final" variable. A blank final instance variable of a class must be definitely assigned in every constructor of the class in which it is declared; similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared; otherwise, a compile-time error occurs in both cases.

Once a final variable has been assigned, it always contains the same value. If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object (this property of final is called non-transitivity). This applies also to arrays, because arrays are objects; if a final variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

5.7 Are Objects Passed by Reference?
In Java, all objects are still passed by value. The value being passed can be an actual value or an address to an object.

From Wikipedia: *In call by value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (frequently by copying the value into a new memory region)*

On the other hand, pass by reference or call by reference is defined as such: *Call by reference (also referred to as pass by reference) is an evaluation strategy where a function receives an implicit reference to a variable used as argument, rather than a copy of its value.*

5.8 Garbage Collection for Java
In Java, any object that is no longer used will be automatically collected away as garbage. Java is able to tell that an object is ready to be deleted should the object not have any references pointing to it.

In the process of Garbage Collection, it duplicates all objects that are still being referred to and changes the address to the new address before freeing up the memory previously used, which includes objects that have been dereferenced.

5.9 Mother of All Classes
Object is the root of all class ancestry. Key methods to take note of are:

- equals(Object obj)
- toString()

5.10 Classes within Classes
Yes, you can write classes within classes. There are four variations of this type of classes:

- Static Nested Classes
- Non-Static Nested Classes or Inner Classes
- Local Inner Classes
- Anonymous Inner Classes

Classes can be nested ad infinitum, e.g. class A can contain class B which contains class C which contains class D, etc. However, more than one level of class nesting is rare, as it is generally bad design.

There are three reasons you might create a nested class:

- **Organisation**: sometimes it seems most sensible to sort a class into the namespace of another class, especially when it won't be used in any other context
- **Access**: nested classes have special access to the variables/fields of their containing classes (precisely which variables/fields depends on the kind of nested class, whether inner or static).
- **Convenience**: having to create a new file for every new type is bothersome, again, especially when the type will only be used in one context

5.10.1 Static Nested Classes

A static class is a class declared as a static member of another class. Just like other static members, such a class is really just a hanger on that uses the containing class as its namespace, e.g. the class Goat declared as a static member of class Rhino in the package pizza is known by the name pizza.Rhino.Goat.

```
package pizza;
public class Rhino {
    // ...
    public static class Goat {
        // ...
    }
}
```

This nested class would be able to access the nesting class' private static members.

5.10.2 Inner Classes

An inner class is a class declared as a non-static member of another class:

```
package pizza;
public class Rhino {
    public class Goat {
    // ...
    }
    private void jerry() {
        Goat g = new Goat();
    }
}
```

Like with a static class, the inner class is known as qualified by its containing class name, pizza.Rhino.Goat, but inside the containing class, it can be known by its simple name. However, every instance of an inner class is tied to a particular instance of its containing class: above, the Goat created in jerry, is implicitly tied to the Rhino instance **this** in jerry.

Otherwise, we make the associated Rhino instance explicit when we instantiate Goat:

```
Rhino rhino = new Rhino();
Rhino.Goat goat = rhino.new Goat();
```

So what does this gain us? Well, the inner class instance has access to the instance members of the containing class instance. These enclosing instance members are referred to inside the inner class via just their simple names:

```
public class Rhino {
    private String barry;
    public class Goat {
        public void colin() {
            System.out.println(barry);
        }
    }
}
```

You can also access the instance members via Rhino.this.barry, as you can access the containing instance as Rhino.this.

5.10.3 Local Inner Classes

A local inner class is a class declared in the body of a method. Such a class is only known within its containing method, so it can only be instantiated and have its members accessed within its containing method. The gain is that a local inner class instance is tied to and can access the final local variables of its containing method. When the instance uses a final local of its containing method, the variable retains the value it held at the time of the instance's creation, even if the variable has gone out of scope (this is effectively Java's crude, limited version of closures).

Because a local inner class is neither the member of a class or package, it is not declared with an access level. (Be clear, however, that its own members have access levels like in a normal class.)

If a local inner class is declared in an instance method, an instantiation of the inner class is tied to the instance held by the containing method's this at the time of the instance's creation, and so the containing class's instance members are accessible like in an instance inner class. A local inner class is instantiated simply via its name, e.g. local inner class Cat is instantiated as new Cat(), not new this.Cat() as you might expect.

An example of a local inner class:

```
void sortName(List<String> names) {
    boolean ascending = true;
    class NameComparator implements Comparator<String> {
        public int compare(String s1, String s2) {
```

```
            if(ascending) {
                return s1.length() - s2.length();
            } else {
                return s2.length() - s1.length();
            }
        }
    }
    ascending = false;
    names.sort(new NameComparator());
}
```

Variable capture plays a huge role here, which will be covered later.

5.10.4 Anonymous Inner Classes

An anonymous inner class is a syntactically convenient way of writing a local inner class. Most commonly, a local inner class is instantiated at most just once each time its containing method is run. It would be nice, then, if we could combine the local inner class definition and its single instantiation into one convenient syntax form, and it would also be nice if we didn't have to think up a name for the class (the fewer unhelpful names your code contains, the better). An anonymous inner class allows both these things:

```
new *ParentClassName*(*constructorArgs*) {*members*}
```

This is an expression returning a new instance of an unnamed class which extends ParentClassName. You cannot supply your own constructor; rather, one is implicitly supplied which simply calls the super constructor, so the arguments supplied must fit the super constructor. (If the parent contains multiple constructors, the **“simplest” one** is called, “simplest” as determined by a rather complex set of rules not worth bothering to learn in detail. Your IDE will tell you.)

Alternatively, you can specify an interface to implement:

```
new *InterfaceName*() {*members*}
```

Such a declaration creates a new instance of an unnamed class which extends Object and implements InterfaceName. Again, you cannot supply your own constructor; in this case, Java implicitly supplies a no-arg, do-nothing constructor (so there will never be constructor arguments in this case).

Even though you can't give an anonymous inner class a constructor, you can still do any setup you want using an initialiser block (a {} block placed outside any method). An example of an initialiser block:

```
public class GFG
{
    // Initializer block starts..
    {
        // This code is executed before every constructor.
        System.out.println("Common part of constructors invoked !!”);
    }
    // Initializer block ends
```

```
}
```

The initialiser block contains the code that is always executed whenever an instance is created. It is used to declare/initialise the common part of various constructors of a class.

Be clear that an anonymous inner class is simply a less flexible way of creating a local inner class with one instance. If you want a local inner class which implements multiple interfaces or which implements interfaces while extending some class other than Object or which specifies its own constructor, you're stuck creating a regular named local inner class.

5.11 Variable Capture
A local class has access to the members of its enclosing class. In addition, a local class has access to local variables. However, a local class can only access **local variables** that are declared final (does not include members of the enclosing class). When a local class accesses a local variable or parameter of the enclosing block, it **captures** that variable or parameter. For example, the PhoneNumber constructor can access the local variable numberLength because it is declared final; numberLength is a captured variable.

However, starting in Java SE 8, a local class can access **local** variables and parameters of the enclosing block that are final or effectively final. A variable or parameter whose value is never changed after it is initialized is effectively final. For example, suppose that the variable numberLength is not declared final, and you add the assignment statement in the PhoneNumber constructor to change the length of a valid phone number to 7 digits:

```
PhoneNumber(String phoneNumber) {
    numberLength = 7;
    String currentNumber = phoneNumber.replaceAll(regularExpression, "");
    if (currentNumber.length() == numberLength)
        formattedPhoneNumber = currentNumber;
    else
        formattedPhoneNumber = null;
}
```

Because of this assignment statement, the variable numberLength is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final" where the inner class PhoneNumber tries to access the numberLength variable:

```
if (currentNumber.length() == numberLength)
```

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters. For example, you can define the following method in the PhoneNumber local class:

```
public void printOriginalNumbers() {
    System.out.println("Original numbers are " + phoneNumber1 + " and " +
phoneNumber2); // where phoneNumber1 and 2 are local variables from the
method
}
```

The method printOriginalNumbers accesses the parameters phoneNumber1 and phoneNumber2 of the method validatePhoneNumber. The rules for variable capture still apply for method parameters.

5.12 Other Things to Take Note
If a local class is defined in a static method, then it can only refer to the static members of the enclosing class.

If a local class is defined in a non-static method:

- They have access to instance members of the enclosing block
- They cannot contain most kinds of static declarations
    - No interfaces, as interfaces are inherently static
    - No static variables or methods or initialisers
    - **However** constants (public static final) are allowed

Chapter 6: Inheritance and Polymorphism

6.1 Interface
An interface is a contract between the two sides of the abstraction barrier. It's a template for a class which lists down how to create the class without any actual implementation.

6.1.1 Default Methods in Interface
Like regular interface methods, default methods are implicitly public — there's no need to specify the public modifier. Unlike regular interface methods, they are declared with the default keyword at the beginning of the method signature, and they provide an implementation.

Default methods were introduced to allow easy expansion of an interface. Prior to its introduction, if any interface needed additional methods, all implementing classes would need an implementation of their own of that method, even if it's a standardised one. As such, backward compatibility is not preserved.

Here is an example of a default method in Interface:

```
public interface Vehicle {
    String getBrand();
    String speedUp();
    String slowDown();
    default String turnAlarmOn() { return "Turning the vehicle alarm on.";
}
    default String turnAlarmOff() { return "Turning the vehicle alarm
off."; }
}
```

And a class that implements it:

```
public class Car implements Vehicle {
    private String brand;
    // constructors/getters
    @Override
    public String getBrand() { return brand; }
    @Override
    public String speedUp() { return "The car is speeding up.”; }
    @Override
    public String slowDown() { return "The car is slowing down.”; }
    public static void main(String[] args) {
        Vehicle car = new Car("BMW");
        System.out.println(car.getBrand()); // BMW
        System.out.println(car.speedUp()); // The car is speeding up.
        System.out.println(car.slowDown()); // The car is slowing down.
        System.out.println(car.turnAlarmOn()); // Turning the vehicle alarm
on.
        System.out.println(car.turnAlarmOff()); // Turning the vehicle
alarm off.
    }
}
```

6.1.2 Static Methods in Interface

Similar to static methods in a class, static methods in an interface will need to be called with the interface name preceding the method name. This static method can be invoked in other static and default methods.

For example:

```
public interface Vehicle {
    // regular / default interface methods
    static int getHorsePower(int rpm, int torque) {
        return (rpm * torque) / 5252;
    }
}

Vehicle.getHorsePower(2500, 480)); // returns horsepower
```

6.1.3 Diamond Problem from Multiple Interface Inheritance

Since a single class can implement multiple interfaces, the diamond problem actually occurs when a class implements several interfaces with the same default methods and method signatures.

Using the above Car example, let’s say we have another Alarm interface:

```
public interface Alarm {
    default String turnAlarmOn() { return "Turning the alarm on."; }
    default String turnAlarmOff() { return "Turning the alarm off."; }
}

public class Car implements Vehicle, Alarm {
```

```
    // ...
}
```

In this case, the code will actually not compile, since there's a conflict caused by multiple interface inheritance. There is thus ambiguity. To solve this ambiguity, we would need to provide an explicit implementation for the methods. It's even possible to use both sets of default methods:

```
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();
}

@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff();
}
```

6.2 Superclass and Subclass

Superclass is also called the parent class. The subclass is also called the child class.

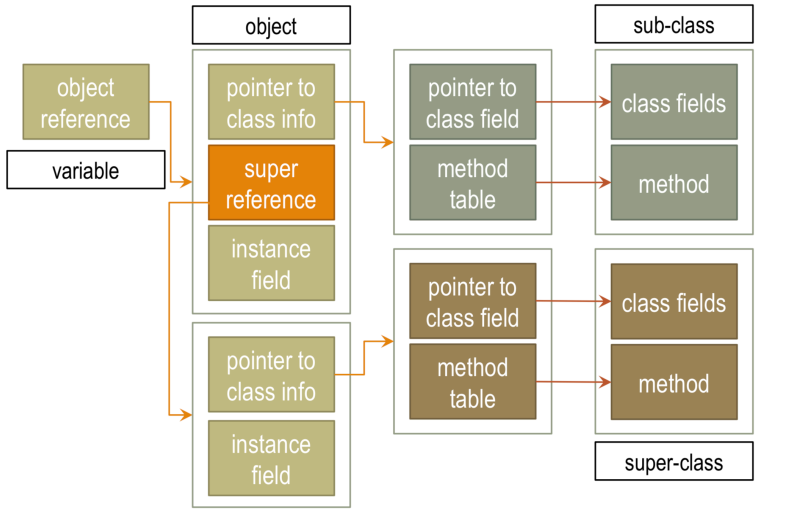
6.3 Quote

Each significant piece of functionality in a program should be implemented in just one place in the source code.

6.4 Multiple Inheritance of Classes?

No.

6.5 Simplified Diagram of the Java Memory Model with Superclasses



6.6 Liskov Substitution Principle

Let $\phi\ x$ be a property provable about object x of type T. Then$\phi\ y$ should be true for object y of type S where S is a subtype of T. If S is a subclass of T, then object of type T can be replaced by an object of type S without changing the desirable property of the

program.

6.7 Instantiation, Early Binding and Late Binding
6.7.1 Context
Shape: interface
Circle: class implements Shape
Square: class implements Shape
Rectangle: class extends Square

6.7.2 Valid Instantiations

```
Square square = new Square();
Square square = new Rectangle(); // Late binding - covariant
Rectangle rect = new Rectangle();
```

6.7.3 Invalid Instantiations

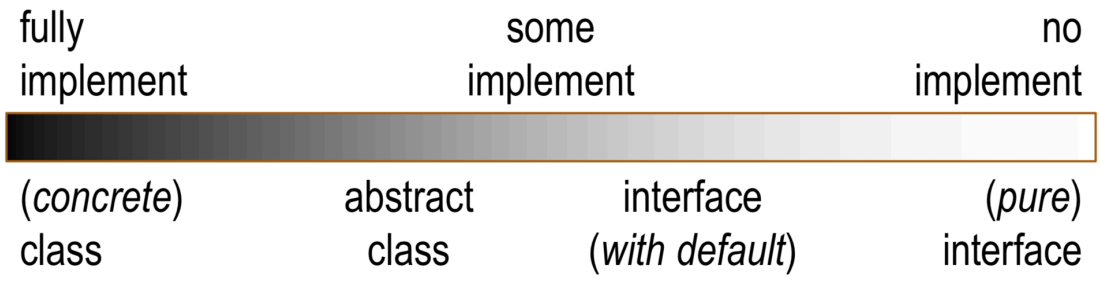
```
Rectangle rect = new Square(); // This is contravariant
```

6.8 Inheritance vs Composition
Inheritance describes an is-a relationship, while composition describes a has-a relationship.

6.9 Abstract Classes
A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

It can also have its own constructors and static methods, as well as final methods which stops subclasses from overriding them.

The key rule is: as long as there is a single abstract method in a class, the class will need to be declared as abstract.



6.10 Polymorphism
The word means “the capacity to take on different forms”.

These are the key points:

- An object in Java that passes more than one IS-A tests is polymorphic in nature
- Every object in Java passes a minimum of two IS-A tests: one for itself and one for Object class
- Static polymorphism in Java is achieved by method overloading
- Dynamic polymorphism in Java is achieved by method overriding

Chapter 7: Types and Subtypes

7.1 Symbolic Execution
Symbolic execution is not just applicable for input types, but also for input values. It's basically a means of analysing what inputs would result in each part of the program to execute.

Not very sure why this was introduced here, but more can be read up at: *https://en.wikipedia.org/wiki/Symbolic_execution*

7.2 All About Subtypes
7.2.1 Subtype Notation
When you say B is a subtype of A, then B<:A.

7.2.2 Transitive Subtyping
If A is a subtype of B and B is a subtype of C, then A is a subtype of C. Also can be represented as:
If A<:B and B<:C then A<:C

7.2.3 Reflexive Subtyping
Any element, A, is a subtype of itself, aka A<:A

7.2.1 Covariance
What is variance? Many programming language type systems support subtyping. Variance refers to how subtyping between more complex types relates to subtyping between their components.

Covariance means that if supertypes are accepted, then subtypes will also be accepted.

Arrays in Java are covariant, hence there are two implications:

1. Firstly, an array of type T[] may contain elements of type T and its subtypes. An array of B thus can contain A inside if A<:B.

```
Number[] numbers = newNumber[3];
numbers[0] = newInteger(10);
numbers[1] = newDouble(3.14);
numbers[2] = newByte(0);
```

2. Secondly, an array of type S[] is a subtype of T[] if S is a subtype of T. This means that if A<:B then A[]<:B[].

```
Integer[] myInts = {1,2,3,4};
```

```
Number[] myNumber = myInts;
```

However, it's important to remember that myNumber is a reference of reference type Number[] to the "actual object" myInts of "actual type" Integer[]. As arrays are of a **reifiable** type, the run-time type system is able to tell that the array was actually instantiated as an array of integers (Integer[]) that simply happens to be accessed through a reference of type Number[].

Therefore, the following line will compile just fine, but will produce a runtime ArrayStoreException (because of heap pollution):

```
myNumber[0] = 1.23; // Not ok
```

It produces a runtime exception, because Java knows at runtime that the "actual object" myInts is actually an array of Integer.

We can also explore Covariance using wildcards, which will be explained further.

7.2.4 Contravariance

Contravariance means that supertypes are accepted. Contravariance in Java will be explored in further detail in wildcards.

7.2.5 Invariance

Invariance means that neither subtypes nor supertypes are accepted - only the exact type will be. In Java, generic types are invariant. With generic types, Java has no way of knowing at runtime the type information of the type parameters, due to type erasure. Therefore, it cannot protect against heap pollution at runtime. To prevent that, they make generic types invariant.

```
ArrayList<Integer> intArrList = new ArrayList<>();
ArrayList<Number> numArrList = intArrList; // Not ok
ArrayList<Integer> anotherIntArrList = intArrList; // Ok
```

Generics affect the power of polymorphism, as a class with parameterised type T is not a subclass of another class with parameterised type E, where T<:E. For example, we cannot consider a List<Integer> to be a subtype of List<Number>, and a method sum(List<Number>) will not be able to operate on lists of Integers, Doubles etc.

To overcome this problem, wildcards are used. This will be explored further later.

7.2.6 Bivariance

Bivariance means that both subtypes and supertypes are accepted. This is demonstrated in Java using an unbounded wildcard on the type parameter. A generic type with an unbounded wildcard is a supertype of all bounded variations of the same generic type. Eg. GenericType<?> is a supertype of GenericType<String>. Since the unbounded type is the root of the type hierarchy, it follows that of its parametric types it can only access methods inherited from java.lang.Object.

```
Think of GenericType<?> as GenericType<Object>.
```

7.3 Primitive Types and Conversions

7.3.1 Widening and Narrowing and Typecasting

When a number is promoted from a subtype that is “smaller” to one that’s “larger”, widening occurs. The reverse is called narrowing, and will throw an error if explicit typecasting is not performed:

```
Error: incompatible types: possible lossy conversion from double to float
```

Truncation occurs when explicit typecasting is done for types such as double, float and integer.

The same terminology applies for usual classes and subclasses.

7.3.2 Autoboxing and Auto-unboxing

Boxing and unboxing is when primitive types are converted to their wrapper class equivalents, or vice versa respectively. For example, we can have:

```
int x = 3;
Integer a = x; // autoboxing
int y = a; // unboxing
```

7.4 Wrapper Class Caching

Basically, the Integer class keeps a cache of Integer instances in the range of -128 to 127, and all autoboxing, literals and uses of Integer.valueOf() will return instances from that cache for the range it covers.

This is based on the assumption that these small values occur much more often than other ints and therefore it makes sense to avoid the overhead of having different objects for every instance (an Integer object takes up something like 12 bytes)

Note that the cache only works if you use auto-boxing or the static method Integer.valueOf(). Calling the constructor will always result in a new instance of integer, even if the value of that instance is in the -128 to 127 range.

Other wrapper classes also have their own cache:

- java.lang.Boolean store two inbuilt instances TRUE and FALSE, and return their reference if new keyword is not used.
- java.lang.Character has a cache for chars between unicodes 0 and 127 (ascii-7 / us-ascii).
- java.lang.Long has a cache for long between -128 to +127.
- java.lang.String has a whole new concept of string pool.

Chapter 8: Generics and Wildcard

8.1 What are Generics?

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively. Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

For example, Pair<T> is the **generic class**. Pair<Sock> is the **parameterised class**, where T is the formal type parameter and Sock is the actual type argument.

Limitations for Generics
You cannot set a variable of a generic type as a static field for a class:

```
|  Error:
|  non-static type variable T cannot be referenced from a static context
|  static T y;
|         ^
```

If you want a static method to return a generic type result, you need to write the method as:

```
static <X> X foo(X t) {
    // ...
}
```

if not you will get the following error:

```
|  Error:
|  non-static type variable T cannot be referenced from a static context
|  static T foo (T t){return t;}
|                ^
|  Error:
|  non-static type variable T cannot be referenced from a static context
|  static T foo (T t){return t;}
|         ^
```

You also cannot use a primitive for a generic type:

```
Pair<int> x = new Pair<>();
|  Error:
|  unexpected type
|    required: reference
|    found:    int
|  Pair<int>
|       ^-^
```

The Diamond
In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the

compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance of Box<Integer> with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

If you type something like this:

```
List<? super Integer> foo3 = new ArrayList<>();
```

It will infer the type to be Integer.

You cannot use it as such:

```
Pair<> p = new Pair<Object, Object>();
```

Because the diamond operator can only be used when instantiating a generic type, not as a declared type.

8.2 Bounded and Unbounded Generics
There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter:

- List the type parameter's name
- Along with the extends keyword
- And its upper bound

```
<T extends superClassName>
```

We can also have multiple bounds with interfaces:

```
<T extends superClassName & Interface>
```

DO NOT say that superClassName extends superClassName, and is thus accepted. Should explain it as superClassName is upper-bounded by ? extends superClassName. The reverse is true for super.

8.3 How does Generics ensure Type Safety?
Generics allow classes and methods that use any reference type to be defined without resorting to using the Object type. It enforces type safety by binding the generic type to a specific given type argument at **compile time**. Attempt to pass in an incompatible type would lead to compilation error.

OR

<> helps compiler in verifying type safety. Compiler makes sure that List<MyObj> holds objects of type MyObj at compile time instead of runtime. Generics are mainly for the

purpose of type safety at compile time. All generic information will be replaced with concrete types after compilation due to type erasure.

8.4 What are Wildcards?
A wildcard is simply the symbol ? which stands for an unknown type.

Consider the following code:

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

This code looks like it will work, except that it doesn't. Only an actual Collection<Object> will be able to be passed in, while any other Collections will be rejected:

```
|  Error:
|  incompatible types: java.util.Collection<java.lang.Integer> cannot be
converted to java.util.Collection<java.lang.Object>
|  printCollection(ic)
|                  ^^
```

So what is a supertype of all Collections? It's written as Collection<?>, read as "collection of unknown", which is a collection whose element type matches **anything.** It's called a wildcard type, as we mentioned above.

Now we can write the function as:

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

And we can now call it with any type of collection. Notice that inside printCollection(), we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects.

However, if we declare a new object with type ?, we cannot just add anything into it.

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Compile time error
c.add(1); // Error
c.add("Hello"); // Error
```

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type. **(This is similar to <? extends Object>)**

On the other hand, given a List<?>, we can call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

8.5 Bounded Wildcards
There are two key words to take note of:

8.5.1 extends
The wildcard declaration of List<? extends Number> foo3 means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends"
Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends
Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends
Number
```

Reading - Given the above possible assignments, what type of object are you guaranteed to read from List foo3:

- You can read a Number because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number.
- You can't read an Integer because foo3 could be pointing at a List<Double>.
- You can't read a Double because foo3 could be pointing at a List<Integer>.

Writing - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for all the above possible ArrayList assignments:

- You can't add an Integer because foo3 could be pointing at a List<Double>.
- You can't add a Double because foo3 could be pointing at a List<Integer>.
- You can't add a Number because foo3 could be pointing at a List<Integer>.

You can't add any object to List<? extends T> because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or subclass of T.

8.5.2 super
Now consider List <? super T>.

The wildcard declaration of List<? super Integer> foo3 means that any of these are legal assignments:

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a
"superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a
superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a
```

```
superclass of Integer
```

Reading - Given the above possible assignments, what type of object are you guaranteed to receive when you read from List foo3:

- You aren't guaranteed an Integer because foo3 could be pointing at a List<Number> or List<Object>.
- You aren't guaranteed a Number because foo3 could be pointing at a List<Object>.
- The only guarantee is that you will get an instance of an Object or subclass of Object (but you don't know what subclass).

Writing - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for all the above possible ArrayList assignments:

- You can add an Integer because an Integer is allowed in any of above lists.
- You can add an instance of a subclass of Integer because an instance of a subclass of Integer is allowed in any of the above lists.
- You can't add a Double because foo3 could be pointing at an ArrayList<Integer>.
- You can't add a Number because foo3 could be pointing at an ArrayList<Integer>.
- You can't add an Object because foo3 could be pointing at an ArrayList<Integer>.

8.5.3 Writing a function that takes in a List of subtypes of Number

There are two ways to write it

```
void printNumbers(List<? extends Number> list){
    for (Number i : list){...}
}

<T extends Number> void printNumbers(List<T> list){
    for (Number i : list){...}
}
```

8.6 PECS

Remember PECS: **"Producer Extends, Consumer Super"**.

- "Producer Extends" - If you need a List to produce T values (you want to read Ts from the list), you need to declare it with ? extends T, e.g. List<? extends Integer>. But you cannot add to this list.
- "Consumer Super" - If you need a List to consume T values (you want to write Ts into the list), you need to declare it with ? super T, e.g. List<? super Integer>. But there are no guarantees what type of object you may read from this list.
- If you need to both read from and write to a list, you need to declare it exactly with no wildcards, e.g. List<Integer>.

8.6.1 Example

Note this example from the Java Generics FAQ. Note how the source list src (the

producing list) uses extends, and the destination list dest (the consuming list) uses super:

```
public class Collections {
    public static <T> void copy(List<? super T> dest, List<? extends T>
src) {
        for (int i = 0; i < src.size(); i++)
            dest.set(i, src.get(i));
    }
}
```

8.7 Variance of Generics and Wildcard

8.7.1 Subtypes within Declared Classes

If T <: S then T<X> <: S<X> (i.e., covariant)

```
class S <X>{}
class T<X> extends S<X>{}
T<Integer> x = new T<>();
S<Integer> y = x; // no error
```

8.7.2 Subtypes within Generics

If T <: S then X<T> and X<S> are invariant, which was what we covered before in both Invariance and Wildcard. For example, a method accepting ArrayList<Object> will not accept an ArrayList<Integer>.

8.7.3 extends

If T <: S then X<T> <: X<? extend S>

```
ArrayList<? extends Number> x = new ArrayList<Integer>(); // no error
```

8.7.4 super

If T <: S then X<S> <: X<? super T>

```
ArrayList<? super Number> y = new ArrayList<Object>();
```

Should the object be a list, then separate rules apply when it comes to subsequent insertion, where instances of Number and its subclasses can also be inserted into the list. For example:

```
y.add(new Integer(1));
y.add(new Double(1));
```

All of the elements inserted here are either Number instances, or instances of Number's superclass. Since both Integer and Double extend Number, if Number had a superclass, Integer and Double would also be instances of that superclass.

8.8 Type Erasure

To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only

ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.
- Type erasure ensures that no new classes are created for parameterised types; consequently, generics incur no runtime overhead.

8.8.1 Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
    // ...
}
```

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```
public class Node {
    private Object data;
    private Node next;
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() { return data; }
    // ...
}
```

In the following example, the generic Node class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

```
public class Node {
    private Comparable data;
    private Node next;
    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Comparable getData() { return data; }
    // ...
}
```

8.8.2 Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```
// Counts the number of occurrences of elem in anArray.
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Because T is unbounded, the Java compiler replaces it with Object:

```
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Suppose the following classes are defined:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces T with Shape:

```
public static void draw(Shape shape) { /* ... */ }
```

8.8.3 Implications

You cannot have two methods with method signatures differentiated by generics:

```
import java.util.List;
class A {
    void foo(List<Integer> integerList) {}
    void foo(List<String> stringList) {}
}

|  Error:
|  name clash: foo(java.util.List<java.lang.String>) and
foo(java.util.List<java.lang.Integer>) have the same erasure
|      void foo(List<String> stringList) {}
|      ^---------------------------------^
```

You cannot instantiate a static variable with generics (more due to the variable nature of generics):

```
class B<T> {
    T x;
    static T y;
}

|  Error:
|  non-static type variable T cannot be referenced from a static context
|      static T y;
|             ^
```

Chapter 9: Exception Handling

9.1 Keywords for Manual Exception Handling

Java Exception Handling is managed via five keywords: **try, catch, throw, throws,** and **finally**. Briefly, here is how they work. Program statements that you think may possibly raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can then catch this exception using the **catch** keyword and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a try block completes is put in a **finally** block.

9.2 How does the Java Virtual Machine handle Exceptions?

Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system (JVM). The exception object contains the name and description of the exception, and the current state of the program where the exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception.

The ordered list of the methods that led to the eventual method which threw this error is

called Call Stack. Now the following procedure will happen:

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred Exception. This block of the code is called Exception Handler.
- The run-time system starts its search from the method in which the Exception occurred, and proceeds through call stack in the reverse order in which methods were called.
- If it finds an appropriate Handler, it will pass the occurred Exception to it. An appropriate handler means that the type of the Exception Object thrown matches the type of the Exception Object the Handler can handle.
- If the run-time system searches all the methods on call stack and couldn't find an appropriate Handler, it will hand over the Exception Object to the Default Exception Handler, which is part of the run-time system. This Handler will print the Exception information in the following format and terminate program abnormally:

```
Exception in thread "xxx" Name of Exception : Description
... ...... ..  // Call Stack
```

9.3 Catching Exceptions

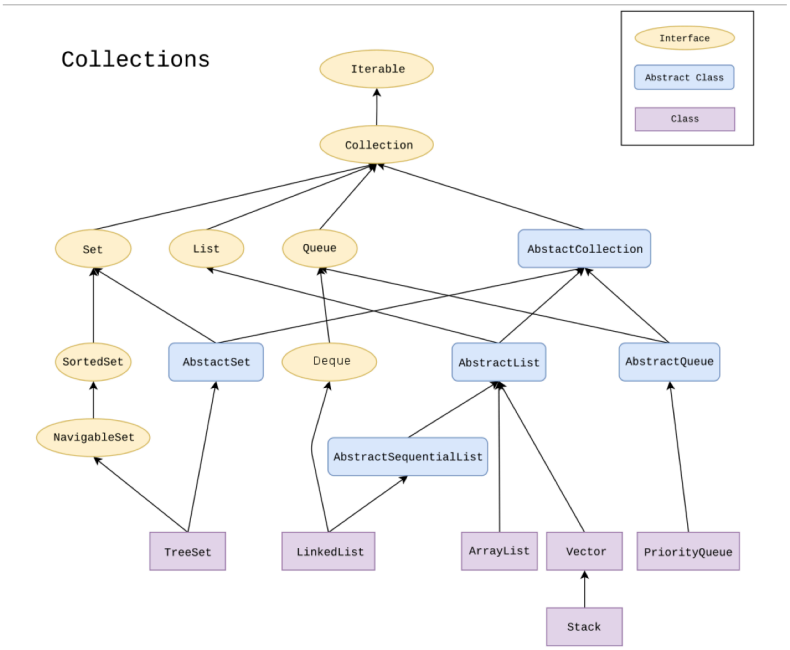
Sample code:

```
public void f() throws Exception{
    try{
        System.out.println("Throwing Exc2...");
        throw new Exc2();
    } catch (Exc1 e){
        System.out.println("Exc1 is caught.");
    }
}
```

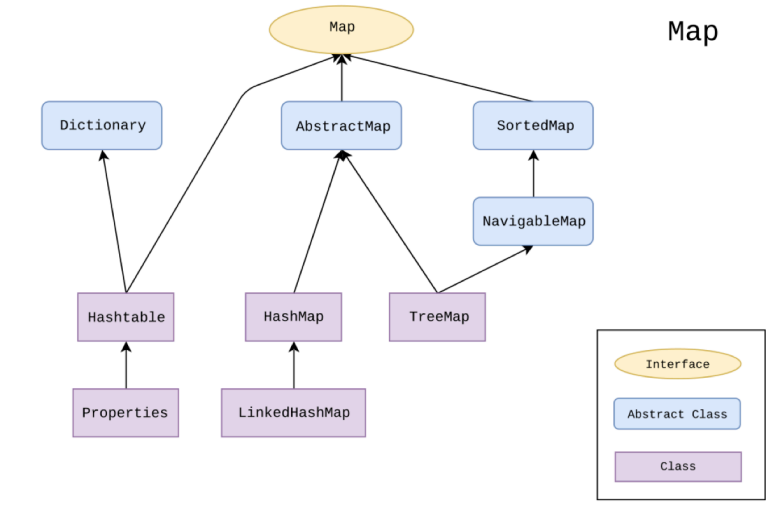
So what can **catch** catch? It can catch the subclasses of the Exception class within the bracket, including the class itself. It, however, cannot catch the parent classes.

Chapter 10: Java API and Packages

10.1 Collections



10.2 Map



10.3 Summary of Various Data Structures

Summary

Stack<E>

Vector<E>

PriorityQueue<E>

HashTable<K,V>

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

https://docs.oracle.com/javase/7/docs/technotes/guides/collections/reference.html

10.4 Equality Check with HashMap
If the value of x is equal to the value of y, then the hash code of x is equal to the hash code of y.
If the hash code x is not equal to the hash code of y, then the value of x is not equal to the value of y.

If one overrides the method equals() from Object, then we must override the method hashCode() as well, as it is required that the two objects x and y satisfy the following Property P:
If x.equals(y) then x.hashCode() == y.hashCode().

It is also equally important that the hashCode() is well-written, else all objects of a certain class will be hashed to the same few buckets in HashSet and HashMap, and searching and retrieving will be highly inefficient as we have to search through all elements every time. We would also end up being unable to use hashCode() to separate two objects that are different in equals.

10.5 Scanner
The keyboard acts as the standard input, which can be read with java.util.Scanner and System.in. The monitor acts as the standard output, where things can be printed using System.out.print, System.out.println, and System.out.printf.

10.5.1 Scanner Functions

- Integer: sc.nextInt(); sc.nextLong(); sc.nextByte(); sc.nextShort(); sc.
- Double: sc.nextDouble(); sc.nextFloat();
- Word: sc.next();
- Line/String/Read \n character: sc.nextLine();

- Boolean: sc.hasNext(); sc.hasNextInt(); sc.hasNextLine(); etc.

10.5.2 File Input and Output

Using Scanner:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ScannerReadFile {
    public static void main(String[] args) {
        File file = new File("data.txt");
        try {
            Scanner scanner = new Scanner(file);
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Using FileReader:

```
import java.io.*;
public class ReadingFromFile {
    public static void main(String[] args) throws Exception {
        FileReader fr = new FileReader("test.txt”);
        int i;
        while ((i=fr.read()) != -1) System.out.print((char) i);
    }
}
```

10.6 Packages

To use a package, we can simply use the keyword import. To import all sub packages within a larger package, we can use *, e.g. import java.util.*

To create a package, we will need to use the keyword package. All files of a package will need to be in the same folder.

```
package cs2030.shapes;
public interface Shape{}
```

To compile all of the java files and have them automatically fall into the appropriate package folders, the following command can be used:

```
javac -d . *.java
```

For example, the Shape.java above will compile into folder cs2030 > shapes >

Shape.class.

10.7 Random
The random generator in Java is a linear congruent generator. The formula for generation is as such:
X(n) = (a×X(n-1) + c) mod m
where X(0) is the seed.

10.7.1 java.util.Random
Mainly Uniform Distribution

- nextInt(): [0, MAX_INT]
- nextInt(int n): [0, n) or [0, n-1]
- nextDouble(): [0.0, 1.0]

To generate a uniform distribution of [5, 12], simply do:

```
Random rand = new Random();
int n = rand.nextInt(8); // [0, 7]
n = n + 5; // [5, 12]
```

Gaussian Distribution

- nextGaussian(): mean = 0.0, std dev = 1.0

Exponential Random Variable

- Let x be a randomly distributed number between [0.0, 1.0]
- Let y be a particular rate e.g. arrival rate
- Then the time between 2 arrivals is t = -log(x)/y

Chapter 11: Enum

11.1 What is Enum?
An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

```
enum Level {
    LOW, MEDIUM, HIGH
}

// Reference with dot operator
Level myVar = Level.MEDIUM;
```

```
// Using it in switch case
public class MyClass {
    public static void main(String[] args) {
        Level myVar = Level.MEDIUM;
        switch(myVar) {
            case LOW:
                System.out.println("Low level");
                break;
            case MEDIUM:
                System.out.println("Medium level");
                break;
            case HIGH:
                System.out.println("High level");
                break;
        }
    }
}

// Iteration through the constants
for (Level myVar : Level.values()) {
    System.out.println(myVar);
}
```

11.2 How does Enum ensure Type Safety?
Enum allows a type to be defined and used for a set of predefined constants. Using a constant other than those predefined would lead to a compilation error. In contrast, using int is not type safe since int values other than those predefined can be accidentally assigned or passed as arguments.

11.3 values(), ordinal() and valueOf() methods
To be expanded