



Core Java Fundamentals

🕒 Created	@January 29, 2022 3:02 PM
☑ Reviewed	<input type="checkbox"/>

Table of contents

[Table of contents](#)

[Java language](#)

[Object-Oriented Programming Concepts](#)

[Language Basics](#)

[Operators](#)

[Control Flow statements](#)

[Class](#)

[Object](#)

[Nested class](#)

[Interface](#)

[Inheritance](#)

[Number](#)

[Character](#)

[String](#)

[Autoboxing and unboxing](#)

[Generics](#)

[Package](#)

Java

▼ Java language

1. Advantages: https://www.youtube.com/watch?v=neaGeFZ2j_w
 - a. sufficiently good control and convince.
 - b. Cross platform support
 - c. large set of built in APIs. Eg. Multithreading, IO, etc
 - d. Large open source + commercial ecosystem
 - e. large global community
 - f. popularity

g. frequent regular update

2. Source code(.java) ---compiler---> Java byte code(.class) ---java virtual machine---> Program executed

The Java Virtual Machine is implemented for several different operating systems, like Windows, Mac OS, Linux, IBM mainframes, Solaris etc. "Write once, run everywhere". Thus, if your Java program can run on a Java Virtual Machine on Windows, it can normally also run on a Java Virtual Machine on Mac OS or Linux.

If you are not a programmer and have the .class file then you just need a JRE to run Java programs. JRE is platform dependent because configuration of each OS differs.

The Java compiler and the JVM are part of the Java Development Kit.

3. **What is compiled and interpreted language?** <https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/>

In a compiled language, the target machine directly translates the program. In an interpreted language, the source code is not directly translated by the target machine. Instead, a different program, aka the interpreter, reads and executes the code.

EXAMPLE: Imagine you have a hummus **recipe** that you want to **make**, but it's written in ancient **Greek**. There are two ways you, a non-ancient-Greek speaker, could follow its directions.

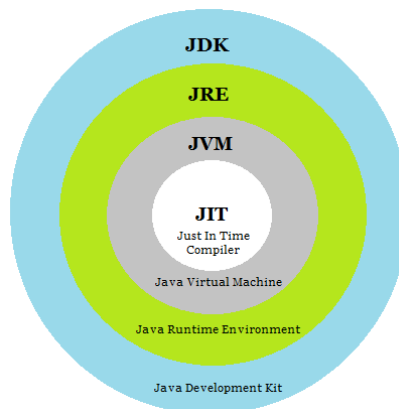
The first is if someone had already translated it **into English** for you. You (and anyone else who can speak English) could read the English version of the recipe and make hummus. Think of this translated recipe as the **compiled version**. Eg. C.

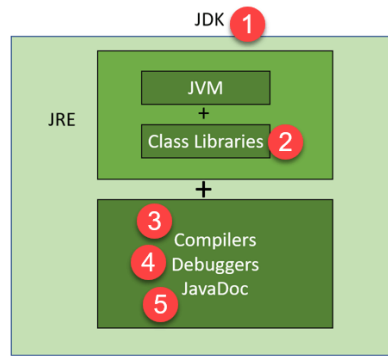
The second way is if you have a friend who knows ancient Greek. When you're ready to make hummus, your **friend** sits next to you and **translates** the recipe into English as you go, **line by line**. In this case, your friend is the interpreter for the interpreted version of the recipe. Eg. python, javascript.

4. **Java is both compiled and interpreted language.**

In Java, programs are not compiled into executable files; they are compiled into bytecode (as discussed earlier), which the JVM (Java Virtual Machine) then interprets / executes at runtime.

5. **JVM vs JRE vs JDK**





1. JDK and JRE: The JDK enables programmers to create core Java programs that can be run by the JRE, which included Java Virtual Machine and the **standard Java APIs** coming with Java Standard Edition (JSE).
2. Class Libraries: It is a group of dynamically loadable libraries that Java program can call at run time.
3. Compilers: It is a Java program that accepts text file of developers and compiles into Java class file. It is the common form of output given by compiler, which contains Java byte code. In Java, the primary compiler is **Javac**.
4. Debuggers: Debugger is a Java program that lets developers test and debug Java programs.
5. JavaDoc: JavaDoc is documentation made by Sun Microsystems for the Java. JavaDoc can be used generating API documentation in HTML file from the source program

Now read from "How JRE functions?" in

<https://www.guru99.com/difference-between-jdk-jre-jvm.html>

What is Bootstrap, Extension and System Class loader?

http://r4r.in/corejava/interview-question-answers/?request_id=1-16&question_id=286

6. Java APIs

The Java language enables you to package components written in the Java language into APIs (Application Programming Interfaces) which can be used by others in their Java applications. The standard Java APIs are available to all Java applications. The standard Java APIs come bundled with the Java Runtime Environment (JRE) or with the Java SDK which also includes a JRE.

Compiling a .java file

- Try `java -version` on cmd. If java not found, jdk environment is not added.
- Run `sudo gedit ~/.bashrc` on cmd then add following lines at the end of the file.

```
export JAVA_HOME="/home/khush/Downloads/jdk"
export PATH=$PATH:$JAVA_HOME/bin
```
- Open the folder in which java file is present using `cd`.
- `javac MyFile.java` (this creates the `MyFile.class` in the same folder)
- `java MyFile` (this will run the program)
- If first line contains package name like `com.java.concurrency` then open the folder in which `com` is present. Now run `java com.java.concurrency.MyFile`

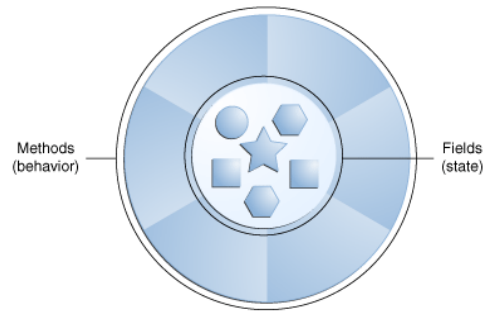
▼ Object-Oriented Programming Concepts

▼ Object

An object is a software bundle of related **state** and **behaviour**.

An object stores its state in *fields* (variables in some programming languages) and exposes its behaviour through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

Hiding internal state and requiring **all interaction to be performed through an object's methods** is known as **data encapsulation** — a fundamental principle of object-oriented programming.



By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity:** The source code for an object can be written and **maintained independently** of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a **bolt breaks, you replace it, not the entire machine**.

▼ Class

A *class* is the blueprint from which individual objects are created.

In object-oriented terms, we say that your bicycle is an **instance** of the *class of objects* known as bicycles.

▼ Code

```
class Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {
```

```

        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" +
            cadence + " speed:" +
            speed + " gear:" + gear);
    }
}

public class BicycleDemo {
    public static void main(String[] args) {

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

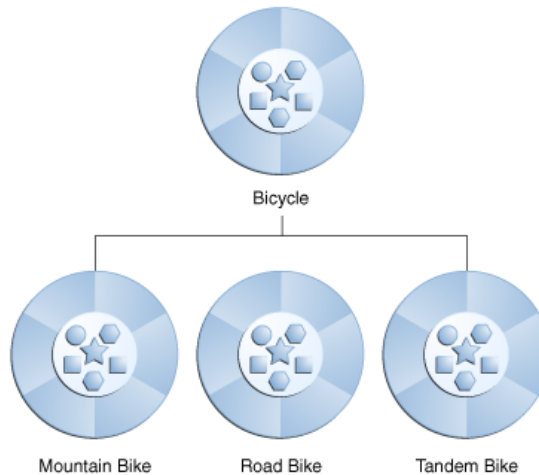
        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}

```

▼ Inheritance

Object-oriented programming allows classes to *inherit* commonly used state and behaviour from other classes. In this example, `Bicycle` now becomes the *super-class* of `MountainBike`, `RoadBike`, and `TandemBike`. In the Java programming language, each class is allowed to have **one direct super-class**, and each super-class has the potential for an **unlimited number of sub-classes**:



The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from:

```
class MountainBike extends Bicycle {  
  
    // new fields and methods defining  
    // a mountain bike would go here  
  
}
```

This gives `MountainBike` all the same fields and methods as `Bicycle`, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

▼ Interface

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, **an interface is a group of related methods with empty bodies**. A bicycle's behaviour, if specified as an interface, might appear as follows:

```
interface Bicycle {  
  
    // wheel revolutions per minute  
    void changeCadence(int newValue);  
    void changeGear(int newValue);  
    void speedUp(int increment);  
    void applyBrakes(int decrement);  
  
}
```

To implement this interface, the name of your class would change, and you'd use the `implements` keyword in the class declaration:

```
class ACMEBicycle implements Bicycle {
```

```

int cadence = 0;
int speed = 0;
int gear = 1;

// The compiler will now require that methods
// changeCadence, changeGear, speedUp, and applyBrakes
// all be implemented. Compilation will fail if those
// methods are missing from this class.

void changeCadence(int newValue) {
    cadence = newValue;
}

void changeGear(int newValue) {
    gear = newValue;
}

void speedUp(int increment) {
    speed = speed + increment;
}

void applyBrakes(int decrement) {
    speed = speed - decrement;
}

void printStates() {
    System.out.println("cadence:" +
        cadence + " speed:" +
        speed + " gear:" + gear);
}
}

```

Implementing an interface allows a class to become more formal about the behaviour it promises to provide.

Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

▼ Package

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "**Application Programming Interface**", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming. For example, a `String` object contains state and behavior for character strings; a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a `Socket` object allows for the creation and use of network sockets; various GUI objects control buttons and check boxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.

The [Java Platform API Specification](#) contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

An `import` statement tells the Java compiler which other Java files this Java file is using. A Java file only needs to import Java files which are not located in the same Java package as the Java file.

▼ Language Basics

▼ Variable

Declaration: [access_modifier] [static] [final] type name [= initial value] ;

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)** Their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.
- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field.

Additionally, the keyword `final` could be added to indicate that the number of gears will never change.
- **Local Variables** They are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters** The `args` variable in `psvm` is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Variable names all are valid and unique:

1. myvar
2. myVar
3. MYVAR
4. _myVar
5. \$myVar

▼ Data types

Each variable in Java has a data type. Data types into two groups:

1. Primitive data types
2. Object references

A variable takes up a certain amount of space in memory. How much memory a variable takes depends on its data type.

A variable of a primitive data type contains the value of the variable directly in the memory allocated to the variable. For instance, a number or a character.

A variable of an object reference type is different from a variable of a primitive type. A variable of an object type is also called a reference. The variable itself does not contain the object, but **contains a reference to the object**.

The reference points to somewhere else in memory where the whole object is stored. Via the reference stored in the variable you can access fields and methods of the referenced object. It is possible to have many different variables reference the same object. This is not possible with primitive data types.

```
Integer myInteger;  
myInteger = new Integer(45);
```


▼ Primitive data type

The **eight** primitive data types supported by the Java programming language are:

- **byte**: The `byte` data type is an **8-bit signed** two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters.
- **short**: The `short` data type is a **16-bit signed** two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.
- **int**: By default, the `int` data type is a **32-bit signed** two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2321. Use the `Integer` class to use `int` data type as an unsigned integer.
- **long**: The `long` data type is a **64-bit** two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$.
- **float**: The `float` data type is a single-precision **32-bit** IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for `byte` and `short`, use a `float` (instead of `double`) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead. Numbers and Strings covers `BigDecimal` and other useful classes provided by the Java platform.
- **double**: The `double` data type is a double-precision **64-bit** IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type **should never be** used for precise values, such as currency.
- **boolean**: The `boolean` data type has only two possible values: `true` and `false`. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined. By default `false`.
- **char**: The `char` data type is a single **16-bit** Unicode character. It has a minimum value of `'\u0000'` (or 0) and a maximum value of `'\uffff'` (or 65,535 inclusive). By default `'\u0000'`.

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the `java.lang.String` class. Enclosing your character string within double quotes will automatically create a new `String` object; for example, `String s = "this is a string";`. **String objects are immutable, which means that once created, their values cannot be changed (need to be referenced with other object as done below).** The `String` class is **not technically a primitive** data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

```
String myString1 = new String("Hello World");
String myString2 = "Hello World";           //same content as above

//Even though the value (text) of the two Java Strings created
//is the same, the Java virtual machine will create two different
//objects in memory to represent them.
```

Default Values

It's not always necessary to assign a value **when a field is declared**. Fields that are declared but not initialized will be **set to a reasonable default by the compiler**.

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. **Accessing an uninitialized local variable will result in a compile-time error.**

Literals

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A *literal* is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

Integer Literals

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise it is of type `int`. It is recommended that you use the upper case letter `L` because the lower case letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed by these number systems:

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix `0x` indicates hexadecimal and `0b` indicates binary:

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type `float` if it ends with the letter `F` or `f`; otherwise its type is `double` and it can optionally end with the letter `D` or `d`. The floating point types (`float` and `double`) can also be expressed using `E` or `e` (for scientific notation), `F` or `f` (32-bit float literal) and `D` or `d` (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

Character and String Literals

The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), and `\\` (backslash).

There's also a special `null` literal that can be used as a value for any reference type. `null` may be assigned to any variable, except variables of primitive types. There's little you can do with a `null` value beyond testing for its presence. Therefore, `null` is often used in programs as a marker to indicate that some object is unavailable.

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (`_`) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code. For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator. The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

▼ Why we need Wrapper Class

- Wrapper Class will **convert primitive data types into objects**. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are **passed by value**).
- The classes in **java.util package** handles only objects and hence **wrapper classes** help in this case also.
- Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object, an address of a system can be seen as an object, an image can be treated as an object (with `java.awt.Image`) and a simple data type can be converted into an object (with wrapper classes). This tutorial discusses wrapper classes. Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced wrapper classes.

▼ Enum type

An *enum type* is a special data type that enables for a variable to be a set of predefined constants.

The enum class body can include methods and other fields. The compiler automatically adds some special methods when it creates an enum. For example, they have a static `values` method that returns an array containing all of the values of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type.

```
public class P4_enum {
    //enum mentioned without access modifier so can be
    //used anywhere inside the package
    enum Size {
        SMALL, MEDIUM, LARGE, X_LARGE
    }

    void method1(Size size){
        switch(size) {

            case SMALL : { System.out.println("size is small"); break; }
            case MEDIUM : { System.out.println("size is medium"); break; }
            case LARGE : { System.out.println("size is large"); break; }
            case X_LARGE : { System.out.println("size is X-large"); break; }

            default : {
                System.out.println("size is not S,M,L or XL: " + size);
            }
        }
    }

    public static void main(String[] str){
```

```

        for (Size s1: Size.values()){ //printing all enum values
            System.out.println(s1);
        }

        P4_enum t1 = new P4_enum();
        t1.method1(Size.MEDIUM);
    }
}
//OUTPUT
SMALL
MEDIUM
LARGE
X_LARGE
size is medium

```

Note: All enums implicitly extend `java.lang.Enum`. Because a class can only extend one parent, the Java language does not support multiple inheritance of state, and therefore an enum cannot extend anything else.

Enum type with parameters

Each enum constant is declared with values for the mass and radius parameters. These values are passed to the constructor when the constant is created. Java requires that the constants be defined first, prior to any fields or methods. Also, when there are fields and methods, the list of enum constants must end with a semicolon.

Note: The constructor for an enum type must be package-private or private access. It automatically creates the constants that are defined at the beginning of the enum body. **You cannot invoke an enum constructor yourself.**

▼ Simple program

```

public class P4_2_enum_parameter {

    enum fix_corners{
        A(1,2),
        B(3,4),
        C(2,2);

        private final double x;
        private final double y;

        fix_corners(double x, double y){
            this.x = x;
            this.y = y;
        }

        double justProduct(double var){
            return var*x*y;
        }
    }

    public static void main(String[] args) {
        for (fix_corners c1: fix_corners.values()){
            System.out.println(c1.justProduct(2));
        }
    }
}
//OUTPUT
4.0
24.0
8.0

```

▼ enum Planets program

In addition to its properties and constructor, `Planet` has methods that allow you to retrieve the surface gravity and weight of an object on each planet. Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);
//after all enums are mentioned use semi colon and don't end bracket

    private final double mass; // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Your weight on %s is %f%n",
                               p, p.surfaceWeight(mass));
    }
}

//run Planet.class from cmd with an argument of 175, output:

$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

▼ Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, **its length is fixed**. If you need an array-like data structure that can change its size, you should use a List.

```
int[] anArray;           //1 way
int myArray[];
```

```

anArray = new int[10];

int[] anArray = {           //2 way
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};

//////////multidimensional array
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {
            {"Mr. ", "Mrs. ", "Ms. "},           //row1
            {"Smith", "Jones"}                   //row2
        };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}

//////////Copying array
//public static void arraycopy(Object src, int srcPos,
//                               Object dest, int destPos, int length)
class ArrayCopyDemo {
    public static void main(String[] args) {
        String[] copyFrom = {
            "a", "b", "c", "d", "e",
            "f", "g", "h", "i", "j", "k",
            "l", "m" };

        String[] copyTo = new String[10];
        System.arraycopy(copyFrom, 2, copyTo, 3, 7);
        for (String coffee : copyTo) {
            System.out.print(coffee + " ");
        }
    }
}
//OUTPUT:
//null null null c d e f g h i

```

Array Manipulations

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the `java.util.Arrays` class. For instance, the previous example can be modified to use the `copyOfRange` method of the `java.util.Arrays` class. The difference is that using the `copyOfRange` method **does not require you to create the destination array** before calling the method, because the destination array is returned by the method:

```

class ArrayCopyOfDemo {
    public static void main(String[] args) {
        String[] copyFrom = {
            "a", "b", "c", "d", "e",
            "f", "g", "h", "i", "j", "k",
            "l", "m" };

        String[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);
        for (String coffee : copyTo) {
            System.out.print(coffee + " ");
        }
    }
}
//OUTPUT:
//c d e f g h i

```

It requires **fewer lines of code**. Note that the second parameter of the `copyOfRange` method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, *exclusively*. In this example, the **range to be copied does not include the array element at index 9** (which contains the string `Lungo`).

Some other useful operations provided by methods in the `java.util.Arrays` class are:

- Searching an array for a specific value to get the index at which it is placed (the `binarySearch` method).

```
int a[] = {1,2,3,5,6};
System.out.print(Arrays.binarySearch(a,4));
//output = -4. There are 3 numbers less than 4 "minus 1(always)".
```

- Filling an array to place a specific value at each index (the `fill` method).
- Sorting an array into ascending order. This can be done either sequentially, using the `sort` method, or concurrently, using the `parallelSort` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.
- Creating a stream that uses an array as its source (the `stream` method). For example, the following statement prints the contents of the `copyTo` array in the same way as in the previous example:

```
java.util.Arrays.stream(copyTo).map(coffee -> coffee + " ").forEach(System.out::print);
```

See [Aggregate Operations](#) for more information about streams.

- Converting an array to a string. The `toString` method converts each element of the array to a string, separates them with commas, then surrounds them with brackets. For example, the following statement converts the `copyTo` array to a string and prints it:

```
System.out.println(java.util.Arrays.toString(copyTo));
```

This statement prints the following:

```
[Cappuccino, Corretto, Cortado, Doppio, Espresso, Frappuccino, Freddo]
```

▼ Summary

The Java programming language uses both "fields" and "variables" as part of its terminology. Instance variables (non-static fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the `static` modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as "variables" (not "fields"). When naming your fields or variables, there are rules and conventions that you should (or must) follow. || The eight primitive data types are: byte, short, int, long, float, double, boolean, and char.

The `java.lang.String` class represents character strings. The compiler will assign a reasonable default value for fields of the above types; for local variables, a default value is never assigned. A literal is the source code representation of a fixed value. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

▼ Annotation

- `@Override`
- `@SuppressWarnings("unchecked")`
- `@Deprecated`
- `@FunctionalInterface`

<https://www.youtube.com/watch?v=JV0atjBcUv4>

▼ Operators

▼ Precedence

When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence	
Operators	Precedence
postfix	<i>expr</i> ++ <i>expr</i> --
unary	++ <i>expr</i> -- <i>expr</i> + <i>expr</i> - <i>expr</i> ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

▼ prefix/postfix unary increment operator

```
public class trial {
    public static void main(String[] args) {
        int i = 3;

        i++;
        System.out.println(i); // prints 4

        ++i;
        System.out.println(i); // prints 5
    }
}
```



```

        System.out.println(++i);// prints 6

        System.out.println(i++);// prints 6

        System.out.println(i);// prints 7
    }
}

```

▼ Conditional operator

The `&&` and `||` operators perform *Conditional-AND* and *Conditional-OR* operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

Conditional operator is `?:`, which can be thought of as shorthand for an `if-then-else` statement. This operator is also known as the **ternary operator** because it uses three operands. Eg. `result = someCondition ? value1 : value2`

▼ instanceof

```

class InstanceofDemo {
    public static void main(String[] args) {

        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}

//Output:
//obj1 instanceof Parent: true
//obj1 instanceof Child: false
//obj1 instanceof MyInterface: false
//obj2 instanceof Parent: true
//obj2 instanceof Child: true
//obj2 instanceof MyInterface: true

```

▼ Bitwise and bit shift operator

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

The bitwise & operator performs a bitwise AND operation.

The bitwise ^ operator performs a bitwise exclusive OR operation.

The bitwise | operator performs a bitwise inclusive OR operation.

The following program, [BitDemo](#), uses the bitwise AND operator to print the number "2" to standard output.

```
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        // prints "2"
        System.out.println(val & bitmask);
    }
}
```

▼ Control Flow statements

▼ Else if vs if

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

        int testScore = 35;

        System.out.println("USAGE OF ELSE IF");
        if(testScore >= 50) {
            System.out.println("Grade = '+'A'");
        } else if(testScore >= 40) {
            System.out.println("Grade = '+'B'");
        } else if(testScore >= 30) {
            System.out.println("Grade = '+'C'");
        } else if(testScore >= 20) {
            System.out.println("Grade = '+'D'");
        } else {
            System.out.println("Grade = '+'F'");
        }

        System.out.println("USAGE OF IF");
        if(testScore >= 50) {
            System.out.println("Grade = '+'A'");
        }
        if(testScore >= 40) {
            System.out.println("Grade = '+'B'");
        }
        if(testScore >= 30) {
            System.out.println("Grade = '+'C'");
        }
        if(testScore >= 20) {
            System.out.println("Grade = '+'D'");
        }
        if(testScore >= 0) {
```

```

        System.out.println("Grade = " + 'F');
    }
}

//OUTPUT
Grade = C
Grade = C
Grade = D
Grade = F

```

▼ Switch statement

A `switch` works with the `byte`, `short`, `char`, and `int` primitive data types. It also works with *enumerated types* (discussed in [Enum Types](#)), the `String` class, and a few special classes that wrap certain primitive types: `Character`, `Byte`, `Short`, and `Integer` (discussed in [Numbers and Strings](#)).

```

public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1: monthString = "January";
                    break;
            case 2: monthString = "February";
                    break;
            case 3: monthString = "March";
                    break;
            case 4: monthString = "April";
                    break;
            case 5: monthString = "May";
                    break;
            case 6: monthString = "June";
                    break;
            case 7: monthString = "July";
                    break;
            case 8: monthString = "August";
                    break;
            case 9: monthString = "September";
                    break;
            case 10: monthString = "October";
                    break;
            case 11: monthString = "November";
                    break;
            case 12: monthString = "December";
                    break;
            default: monthString = "Invalid month";
                    break;
        }
        System.out.println(monthString);
    }
}

//OUTPUT
//August

```

String in Switch expression: The `String` in the `switch` expression is compared with the expressions associated with each `case` label as if the `String.equals` method were being used. In order for the `StringSwitchDemo` example to accept any month regardless of case, `month` is converted to lowercase (with the `toLowerCase` method), and all the strings associated with the `case` labels are in lowercase.

Note: This example checks if the expression in the `switch` statement is `null`. Ensure that the expression in any `switch` statement is not null to prevent a `NullPointerException` from being thrown.

▼ Loops

```
Infinite loop:
while(true) { ... }
if( ; ; ) { ... }

Enhanced for loop:
int myArray[] = {3,2,5,2,3,8};
for( int value : myArray ) {
    sout(value);
}

class BreakWithLabelDemo {           //similar label can be done with CONTINUE
    public static void main(String[] args) {

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

        search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}

//This is the output of the program:
Found 12 at 1, 0
```

▼ Class

▼ Access modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

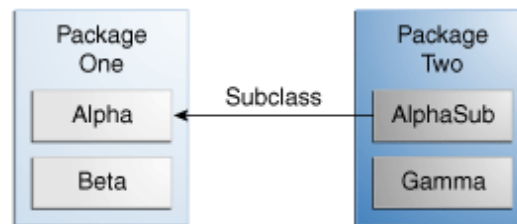
Tips on Choosing an Access Level:

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid `public` fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.

Classes and Packages of the Example Used to Illustrate Access Levels



The following table shows where the **members of the Alpha class** are visible for each of the access modifiers that can be applied to them.

Modifier	Alpha	Beta	Alphasub	Gamma
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
<code>private</code>	Y	N	N	N

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```

public class Bicycle {

    private int gear;
    private int cadence;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getGear() {
        return gear;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getCadence() {
        return cadence;
    }
    public void setCadence(int newValue) {
        cadence = newValue;
    }
}
  
```

```

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}

```

Class Access Modifiers

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers private and protected cannot be assigned to a class. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

▼ Method overloading

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists.

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, `draw(String s)` and `draw(int i)` are distinct and unique methods because they require different argument types.

The compiler does not consider return type when differentiating methods, so you **cannot declare two methods with the same signature even if they have a different return type**.

▼ Constructor

Overloading

```

public class trial {
    public static void main(String[] args) {
        abc a1;
        a1 = new abc(1.0, 2.0);
        a1 = new abc(1, 2);
    }
}

class abc {
    abc(int a, int b) {
        System.out.println("a");
    }
    abc(double a, double b) {
        System.out.println("b");
    }
}

//OUTPUT
//b
//a

```

Constructor is not passed

You **don't have to provide any constructors** for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This **default constructor will call the no-argument constructor of the superclass**. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of `Object`, which *does* have a no-argument constructor.

```
public class trial {
    public static void main(String[] args) {
        childClass c1 = new childClass();
    }
}

class superClass{
    superClass(){
        System.out.println("Super class constructor with no argument called.");
    }
}

class childClass extends superClass{

}

//OUTPUT
//Super class constructor with no argument called.
```

▼ Passing information to method or constructor

▼ Arbitrary Number of Arguments

You can use a construct called **varargs** to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually.

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
        * (corners[1].x - corners[0].x)
        + (corners[1].y - corners[0].y)
        * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // more method body code follows that creates and returns a
    // polygon connecting the Points
}
```

You can see that, inside the method, `corners` is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this `printf` method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s\n", name, idnum, address, phone, email);
```

or with yet a different number of arguments.

▼ Passing Primitive Data Type Arguments

Primitive arguments, such as an `int` or a `double`, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
public class PassPrimitiveByValue {  
  
    public static void main(String[] args) {  
  
        int x = 3;  
  
        // invoke passMethod() with  
        // x as argument  
        passMethod(x);  
  
        // print x to see if its  
        // value has changed  
        System.out.println("After invoking passMethod, x = " + x);  
  
    }  
  
    // change parameter in passMethod()  
    public static void passMethod(int p) {  
        p = 10;  
    }  
}
```

When you run this program, the output is:

```
After invoking passMethod, x = 3
```

▼ Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves `Circle` objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {  
    // code to move origin of circle to x+deltaX, y+deltaY  
    circle.setX(circle.getX() + deltaX);  
    circle.setY(circle.getY() + deltaY);  
  
    // code to assign a new reference to circle  
    circle = new Circle(0, 0);  
}
```

Let the method be invoked with these arguments:

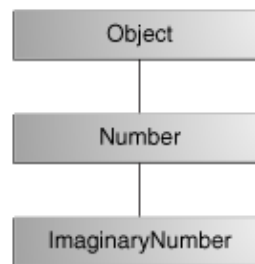
```
moveCircle(myCircle, 23, 56)
```

Inside the method, `circle` initially refers to `myCircle`. The method changes the x and y coordinates of the object that `circle` references (that is, `myCircle`) by 23 and 56, respectively. These changes will persist when the method returns. Then `circle` is assigned a reference to a new `Circle` object with `x = y = 0`. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by `circle` has changed, but, when the method returns, `myCircle` still references the same `Circle` object as before the method was called.

▼ Returning a value

When a method uses a class name as its return type, such as `whosFastest` does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which `ImaginaryNumber` is a subclass of `java.lang.Number`, which is in turn a subclass of `Object`, as illustrated in the following figure.

The class hierarchy for `ImaginaryNumber`



Now suppose that you have a method declared to return a `Number`:

```
public Number returnANumber() {  
    ...  
}
```

The `returnANumber` method can return an `ImaginaryNumber` but not an `Object`. `ImaginaryNumber` is a `Number` because it's a subclass of `Number`. However, an `Object` is not necessarily a `Number` — it could be a `String` or another type.

You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() {  
    ...  
}
```

This technique, called *covariant return type*, means that the return type is allowed to vary in the same direction as the subclass.

Note: You also can use interface names as return types. In this case, the object returned must implement the specified interface.

▼ Using this as constructor

From within a constructor, you can also use the `this` keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*. Here's another `Rectangle` class, with a different implementation from the one in the [Objects](#) section.

```
public class Rectangle {  
    private int x, y;  
    private int width, height;  
  
    public Rectangle() {  
        this(0, 0, 1, 1);  
    }  
}
```

```

public Rectangle(int width, int height) {
    this(0, 0, width, height);
}
public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
...
}

```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 `Rectangle` at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates. As before, the compiler determines which constructor to call, based on the number and the type of arguments.

If present, the invocation of another constructor must be the first line in the constructor.

▼ Order of execution

Order of execution of Initialization blocks and Constructors in Java (refer to my program *P1_order_execution*):

```

public class P1_order_execution {

    //constructor
    P1_order_execution() {
        System.out.println("Constructor block");
    }

    //this is instance initializer block
    {
        System.out.println("IIB block");
    }

    static {
        System.out.println("Static block");
    }

    public static void main(String[] args) {
        System.out.println("Working in main\n");

        P1_order_execution object1 = new P1_order_execution();
        P1_order_execution object2 = new P1_order_execution();

    }
}

//OUTPUT
Static block
Working in main

IIB block
Constructor block
IIB block
Constructor block

```

Order of execution of Initialization blocks and Constructors in Java - GeeksforGeeks

Prerequisite : Static blocks, Initializer block, Constructor In a Java program, operations can be performed on methods, constructors and initialization blocks. Instance Initialization Blocks : IIB are used to initialize instance variables. IIBs are executed before constructors. They run each

<https://www.geeksforgeeks.org/order-execution-initialization-blocks-constructors-java/>



Instance Initialization Blocks : IIB are used to initialize instance variables. IIBs are executed before constructors. They run each time when object of the class is created. ... It is used to declare/initialize the common part of various constructors of a class. Class initializers can also be static. Then they are executed already when the class is loaded, and only once because the class is only loaded in the Java Virtual Machine once.

1. **Static** initialization blocks will run whenever the class is loaded first time in JVM by the classloader
2. Main method
3. Initialization blocks run in the same order in which they appear in the program.
4. Instance Initialization blocks are executed whenever the class is initialized and before constructors are invoked. They are typically placed above the constructors **within braces**.
5. Constructor when object created

Main class: You can have as many classes as you want in your project with a main() method in. But, the Java Virtual Machine can only be instructed to run one of them at a time. You can still call the other main() methods from inside the main() method the Java Virtual Machine executes (you haven't seen how yet) and you can also start up multiple virtual machines which each execute a single main() method.

▼ Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

```
public class BedAndBreakfast {  
  
    // initialize to 10  
    public static int capacity = 10;  
  
    // initialize to false  
    private boolean full = false;  
}
```

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a `for` loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

Note:

It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

Static Initialization Blocks

A *static initialization block* is a normal block of code enclosed in braces, `{ }`, and preceded by the `static` keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks — you can write a private static method:

```
class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {

        // initialization code goes here
    }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the `static` keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

A *final method* cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

▼ Object

▼ Referencing an Object's Fields

You can use any expression that returns an object reference. Recall that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

```
int height = new Rectangle().height;
```

This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

▼ Calling an object's method

Some methods, such as `getArea()`, return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by `getArea()` to the variable `areaOfRectangle`:

```
int areaOfRectangle = new Rectangle(100, 50).getArea();
```

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, **the object that `getArea()` is invoked on is the rectangle returned by the constructor.**

▼ The garbage collector

The Java runtime environment deletes objects when it determines that they are **no longer being** used. This process is called *garbage collection*.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can **explicitly drop an object reference** by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

▼ Nested class

▼ Introduction

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

Inner class is different from the child class because here the nested class has the freedom to use all the variable of the outer class while in case of child class the fields of the parent class need to be public or protected to access them.

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- **It is a way of logically grouping classes that are only used in one place:** If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation:** Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared `private`. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code:** Nesting small classes within top-level classes places the code closer to where it is used.

Nested classes are divided into two categories: non-static and static. Non-static nested classes are called *inner classes*. Nested classes that are declared `static` are called *static nested classes*.

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

▼ Inner classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of `InnerClass` can exist only within an instance of `OuterClass` and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

There are two special kinds of inner classes: local classes and anonymous classes.

▼ Static nested class

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class:

it can use them only through **an object reference of outer class**. [Inner Class and Nested Static Class Example](#) demonstrates this.

Note:

A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

[Inner Class and Nested Static Class Example](#) also demonstrates this.

You instantiate a static nested class the same way as a top-level class:

```
StaticNestedClass staticNestedObject = new StaticNestedClass();
```

▼ Inner Class and Nested Static Class Example

The following example, `OuterClass`, along with `TopLevelClass`, demonstrates which class members of `OuterClass` an inner class (`InnerClass`), a nested static class (`StaticNestedClass`), and a top-level class (`TopLevelClass`) can access:

OuterClass.java

```
public class OuterClass {

    String outerField = "Outer field";
    static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
            // Compiler error: Cannot make a static reference to the non
            // static field outerField
            // System.out.println(outerField);
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }

    public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("-----");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("-----");
        StaticNestedClass staticNestedObject = new StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("-----");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);    //you don't need to
        // create object to access the methods of static nested class
    }
}
```

TopLevelClass.java

```
public class TopLevelClass {

    void accessMembers(OuterClass outer) {
        // Compiler error: Cannot make a static reference to the non-static
        // field OuterClass.outerField
        // System.out.println(OuterClass.outerField);
        System.out.println(outer.outerField);
        System.out.println(OuterClass.staticOuterField);
    }
}
```

This example prints the following output:

```
Inner class:
-----
Outer field
Static outer field

Static nested class:
-----
Outer field
Static outer field

Top-level class:
-----
Outer field
Static outer field
```

Note that a static nested class interacts with the instance members of its outer class just like any other top-level class. The static nested class `StaticNestedClass` can't directly access `outerField` because it's an instance variable of the enclosing class, `OuterClass`. The Java compiler generates an error at the highlighted statement:

```
static class StaticNestedClass {
    void accessMembers(OuterClass outer) {
        // Compiler error: Cannot make a static reference to the non-static
        // field outerField
        System.out.println(outerField);
    }
}
```

To fix this error, access `outerField` through an object reference:

```
System.out.println(outer.outerField);
```

Similarly, the top-level class `TopLevelClass` can't directly access `outerField` either.

▼ Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the **same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope**. You cannot refer to a shadowed declaration by its name alone. The following example, `ShadowTest`, demonstrates this:

```
public class ShadowTest {

    public int x = 0;
```



```

class FirstLevel {

    public int x = 1;

    void methodInFirstLevel(int x) {
        System.out.println("x = " + x);
        System.out.println("this.x = " + this.x);
        System.out.println("ShadowTest.this.x = "+ShadowTest.this.x);
    }
}

public static void main(String... args) {
    ShadowTest st = new ShadowTest();
    ShadowTest.FirstLevel fl = st.new FirstLevel();
    fl.methodInFirstLevel(23);
}
}

```

The following is the output of this example:

```

x = 23
this.x = 1
ShadowTest.this.x = 0

```

This example defines three variables named `x`: the member variable of the class `ShadowTest`, the member variable of the inner class `FirstLevel`, and the parameter in the method `methodInFirstLevel`. The variable `x` defined as a parameter of the method `methodInFirstLevel` shadows the variable of the inner class `FirstLevel`. Consequently, when you use the variable `x` in the method `methodInFirstLevel`, it refers to the method parameter. To refer to the member variable of the inner class `FirstLevel`, use the keyword `this` to represent the enclosing scope:

```

System.out.println("this.x = " + this.x);

```

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class `ShadowTest` from the method `methodInFirstLevel`:

```

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

```

▼ Local and Anonymous Classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. These classes are known as local classes. You can also declare an inner class within the body of a method without naming the class. These classes are known as anonymous classes.

Local Classes

Local classes are classes that are defined in a block, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method. This section covers the following topics: You can define a local class inside any block (see Expressions, Statements, and Blocks for more information).

 <https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html>

Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

 <https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html>

```

package com.java.languagebasics;

class localInner
{
    private int data = 30;//instance variable

    void display()
    {
        class Local
        {
            void msg()
            {
                System.out.println(data);
            }
        }

        Local l = new Local();

        l.msg();
    }
}

abstract class Person
{
    abstract void eat();
}

public class LocalAndAnonymousInnerClass
{
    public static void main(String[] args)
    {
        try
        {
            localInner obj = new localInner(); //local inner class

            obj.display();

            Person p=new Person()           //anonymous inner class
            {
                void eat()
                {
                    System.out.println("nice fruits");
                }
            };

            p.eat();

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}

```

▼ Lambda expression

The previous section, [Anonymous Classes](#), shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. **Lambda expressions let you express instances of single-method classes more compactly.**

To process events in a graphical user interface (GUI) application, such as keyboard actions, mouse actions, and scroll actions, you typically create event handlers, which usually involves implementing a particular interface.

Often, event handler interfaces are **functional interfaces; they tend to have only one abstract method but can have multiple default methods**.

In the JavaFX example `HelloWorld.java` (discussed in the previous section [Anonymous Classes](#)), you can replace the highlighted anonymous class with a lambda expression in this statement:

```
btn.setOnAction(new EventHandler<ActionEvent>() {  
  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Hello World!");  
    }  
});
```

The method invocation `btn.setOnAction` specifies what happens when you select the button represented by the `btn` object. This method requires an object of type `EventHandler<ActionEvent>`.

The `EventHandler<ActionEvent>` interface contains only one method, `void handle(T event)`. This interface is a **functional interface, so you could use the following highlighted lambda expression** to replace it:

```
btn.setOnAction(  
    event -> System.out.println("Hello World!")  
);
```

▼ Syntax of Lambda Expressions

A lambda expression consists of the following:

- A comma-separated list of formal parameters enclosed in parentheses. The `CheckPerson.test` method contains one parameter, `p`, which represents an instance of the `Person` class.

Note: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:

```
p -> p.getGender() == Person.Sex.MALE  
    && p.getAge() >= 18  
    && p.getAge() <= 25
```

- The arrow token, `->`
- A body, which consists of a single expression or a statement block. This example uses the following expression:

```
p.getGender() == Person.Sex.MALE  
    && p.getAge() >= 18  
    && p.getAge() <= 25
```

If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

```
p -> {  
    return p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18  
        && p.getAge() <= 25;  
}
```

A return statement is not an expression; in a lambda expression, you must enclose statements in braces (`{ }`). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

```
email -> System.out.println(email)
```

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name.

The following example, `calculator`, is an example of lambda expressions that take more than one formal parameter:

```
public class Calculator {  
  
    interface IntegerMath {  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    }  
  
    public static void main(String... args) {  
  
        Calculator myApp = new Calculator();  
        IntegerMath addition = (a, b) -> a + b;  
        IntegerMath subtraction = (a, b) -> a - b;  
        System.out.println("40 + 2 = " +  
            myApp.operateBinary(40, 2, addition));  
        System.out.println("20 - 10 = " +  
            myApp.operateBinary(20, 10, subtraction));  
    }  
}
```

The method `operateBinary` performs a mathematical operation on two integer operands. The operation itself is specified by an instance of `IntegerMath`. The example defines two operations with lambda expressions, `addition` and `subtraction`. The example prints the following:

```
40 + 2 = 42  
20 - 10 = 10
```

▼ Program with multiple methods in interface

```
package com.java.languagebasics;  
  
public class LambdaMultipleMethodInterface  
{  
  
    public static void main(String[] Args)  
    {  
        try  
        {  
            myFunction func = (atext, btext) ->  
            {  
                System.out.println("Inside function");  
  
                return atext + " " + btext;  
            };  
  
            String str = func.return_fun("Hey", "Guys");  
        }  
    }  
}
```

```

        System.out.println(str);

        myFunction funcNew = func;

        System.out.println(funcNew.return_fun("Hi", "people"));

        func.my_fun();
    }

    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}

//functional interface = interface having only one abstract method but can have default methods.
//java lambda expression can only implement single method interface i.e. functional interface. Else it gets confused.
interface myFunction
{
    String return_fun(String text1, String text2);

    default void my_fun()
    {
        System.out.println("The default Method");
    }
}

```

▼ Method References

You use [lambda expressions](#) to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

```

public class Person {

    // ...

    LocalDate birthday;

    public int getAge() {
        // ...
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }

    // ...
}

```

Suppose that the members of your social networking application are contained in an array, and you want to sort the array by age. You could use the following code (find the code excerpts described in this section in the example [MethodReferencesTest](#)):

```

Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);

class PersonAgeComparator implements Comparator<Person> {
    public int compare(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}

```

```

    }
}

Arrays.sort(rosterAsArray, new PersonAgeComparator());

```

The method signature of this invocation of `sort` is the following:

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

Notice that the interface `Comparator` is a functional interface. Therefore, you could use a lambda expression instead of defining and then creating a new instance of a class that implements `Comparator`:

```

Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);

```

However, this method to compare the birth dates of two `Person` instances already exists as `Person.compareByAge`. You can invoke this method instead in the body of the lambda expression:

```

Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a, b)
);

```

Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

The method reference `Person::compareByAge` is semantically the same as the lambda expression `(a, b) -> Person.compareByAge(a, b)`. Each has the following characteristics:

- Its formal parameter list is copied from `Comparator<Person>.compare`, which is `(Person, Person)`.
- Its body calls the method `Person.compareByAge`.

Kinds of Method References

There are four kinds of method references:

Kind	Syntax	Examples
Reference to a static method	<code>ContainingClass :: staticMethodName</code>	<code>Person::compareByAge</code> <code>MethodReferencesExamples::appendStrings</code>
Reference to an instance method of a particular object	<code>containingObject :: instanceMethodName</code>	<code>myComparisonProvider::compareByName</code> <code>myApp::appendStrings2</code>
Reference to an instance method of an arbitrary object of a particular type	<code>ContainingType :: methodName</code>	<code>String::compareToIgnoreCase</code> <code>String::concat</code>
Reference to a constructor	<code>ClassName :: new</code>	<code>HashSet::new</code>

▼ Usage

As mentioned in the section [Nested Classes](#), nested classes enable you to logically group classes that are only used in one place, increase the use of encapsulation, and create more readable and maintainable code. Local classes, anonymous classes, and lambda expressions also impart these advantages; however, they are intended to be used for more specific situations:

- **Local class:** Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).
- **Anonymous class:** Use it if you need to declare fields or additional methods.
- **Lambda expression:**
 - Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
 - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).
- **Nested class:** Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.
 - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

▼ Interface

▼ Introduction

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {  
  
    // constant declarations  
  
    // base of natural logarithms  
    double E = 2.718282;  
  
    // method signatures  
    void doSomething (int i, double x);  
    int doSomethingElse(String s);  
}
```

The `public` access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class subclass or extend another class. However, **whereas a class can extend only one other class, an interface can extend any number of interfaces**. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The interface body

The interface body can contain:

- An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation).
- Default methods are defined with the `default` modifier
- Static methods with the `static` keyword.

All abstract, default, and static **methods** in an interface are implicitly `public`, so you can omit the `public` modifier. In addition, an interface can contain constant declarations. All **constant values** defined in an interface are implicitly `public`, `static`, and `final`. Once again, you can omit these modifiers.

Using an Interface as a Type

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

▼ Evolving interfaces (default & static method)

Consider an interface that you have developed called `DoIt`:

```
public interface DoIt {
    void doSomething(int i, double x);
    int doSomethingElse(String s);
}
```

Suppose that, at a later time, you want to add a third method to `DoIt`, so that the interface now becomes:

```
public interface DoIt {

    void doSomething(int i, double x);
    int doSomethingElse(String s);
    boolean didItWork(int i, double x, String s);
}
```

If you make this change, then all classes that implement the old `DoIt` interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and specify it completely from the beginning. If you want to **add additional methods to an interface, you have several options**. You could create a `DoItPlus` interface that extends `DoIt`:

```
public interface DoItPlus extends DoIt {

    boolean didItWork(int i, double x, String s);
}
```

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Alternatively, you can define your new methods as default methods. The following example defines a default method named `didItWork`:

```
public interface DoIt {

    void doSomething(int i, double x);
    int doSomethingElse(String s);
    boolean didItWork(int i, double x, String s);
}
```



```

    default boolean didItWork(int i, double x, String s) {
        // Method body
    }
}

```

Note that you must provide an implementation for **default methods in interface**. You could also **define new static methods** to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

▼ Default methods

The section [Interfaces](#) describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars? These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? *If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.*

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary **compatibility with code written for older versions** of those interfaces.

▼ Extending interfaces that contain default methods

When you extend an interface that contains a default method, you can do the following:

- Not mention the default method at all, which lets your extended interface inherit the default method.
- **Redeclare the default method, which makes it `abstract`.**
- Redefine the default method, which overrides it.

Suppose that you extend the interface `TimeClient` as follows:

```

public interface AnotherTimeClient extends TimeClient { }

```

Any class that implements the interface `AnotherTimeClient` will have the implementation specified by the default method `TimeClient.getZonedDateTime`.

Suppose that you extend the interface `TimeClient` as follows:

```

public interface AbstractZoneTimeClient extends TimeClient {
    public ZonedDateTime getZonedDateTime(String zoneString);
}

```

Any class that implements the interface `AbstractZoneTimeClient` **will have to implement** the method `getZonedDateTime`; this method is an `abstract` method like all other non-default (and non-static) methods in an interface.

Suppose that you extend the interface `TimeClient` as follows:

```

public interface HandleInvalidTimeZoneClient extends TimeClient {
    default public ZonedDateTime getZonedDateTime(String zoneString) {
        try {
            return ZonedDateTime.of(getLocalDateTime(), ZoneId.of(zoneString));
        } catch (DateTimeException e) {

```

```

        System.err.println("Invalid zone ID: " + zoneString +
            "; using the default time zone instead.");
        return ZonedDateTime.of(getLocalDateTime(), ZoneId.systemDefault());
    }
}

```

Any class that implements the interface `HandleInvalidTimeZoneClient` will use the implementation of `getZonedDateTime` specified by this interface instead of the one specified by the interface `TimeClient`.

▼ Static methods in interface

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize **helper methods in your libraries**; you can keep static methods specific to an interface in the same interface rather than in a separate class.

Like static methods in classes, you specify that a method definition in an interface is a static method with the `static` keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly `public`, so you can omit the `public` modifier.

▼ Comparable and Comparator

Comparable (interface)	A comparable object is capable of comparing itself with another object. It is used to order the objects of the user-defined class. This interface contains only one method named <code>compareTo(Object)</code> .
<code>compareTo</code>	The <code>compareTo()</code> means comparison of object itself with ANOTHER. Eg. it compares two strings lexicographically. We can also override that method to compare any object.
Comparator (interface)	Unlike Comparable, Comparator is external to the element type we are comparing . We create multiple separate classes (that implement Comparator) to compare by different members.
<code>compare</code>	Collections class has a <code>sort()</code> method which takes Comparator. The <code>sort()</code> method invokes the <code>compare()</code> to sort objects. <code>compare()</code> means comparison of 2 DIFFERENT objects.

▼ Comparable code

```

package com.java.languagebasics;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

//implementing comparable interface of type student
class student implements Comparable<student>
{
    private int roll;

    private String name;

    student(int r, String s)
    {
        this.roll=r;

        this.name = s;
    }

    public String getName()
    {
        return name;
    }
}

```

```

    }

    public int getRoll()
    {
        return roll;
    }

    //compareTo means comparison of object itself with ANOTHER.
    //compare in comparator means comparison of 2 different object
    @Override
    public int compareTo(student o)        //positive means this is bigger
    {
        return this.roll-o.roll;

        //        return this.name.compareTo(o.name);    //ONLY ONE AT A TIME.

        //        LIMITATION OF Comparable. So, comparator would be used so that both
        //        roll and name can be used together to sort.
    }

    @Override
    public String toString()
    {
        return "student{" +
            "roll=" + roll +
            ", name='" + name + '\'' +
            '}';
    }
}

public class ComparableStudentSorting
{
    public static void main(String[] args)
    {
        List<student> stud = new ArrayList<>();

        try
        {
            stud.add(new student(3, "abc"));

            stud.add(new student(1, "def"));

            stud.add(new student(2, "xyz"));

            student s1 = new student(3, "abc");

            student s2 = new student(1, "def");

            System.out.println("Student 1 and 2 compareTo result: "+s1.compareTo(s2));

            System.out.println("Unsorted:");

            System.out.println(stud);

            System.out.println("Sorted:");

            Collections.sort(stud);

            System.out.println(stud);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}

```

▼ Comparator code

Best one to understand the **different ways** of passing a comparator.

```
package com.java.languagebasics;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class ComparatorStudentSorting
{
    public static void main(String[] args)
    {
        try {

            List<student> studentList = new ArrayList<>();

            studentList.add(new student(3, "def"));

            studentList.add(new student(1, "xyz"));

            studentList.add(new student(1, "fgh"));

            studentList.add(new student(2, "abc"));

            System.out.println("Unsorted:");

            System.out.println(studentList);

            System.out.println("Sorted based on name:");

//            WAY 1
//            Passing comparator using anonymous class

            Collections.sort(studentList, new Comparator<student>()
            {
                @Override
                public int compare(student o1, student o2)
                {
                    return o1.getName().compareTo(o2.getName());
                }
            });

//            WAY 2
//            LAMBDA makes that short in following way
//            Collections.sort(studentList, (o1, o2) -> o1.getName().compareTo(o2.getName()));

//            WAY 3
//            passing comparator with lambda which compares based on name
//            Collections.sort(studentList, Comparator.comparing(student -> student.getName()));

//            WAY 4
//            passing comparator with method reference which compares based on name
//            Collections.sort(studentList, Comparator.comparing(student::getName));

            System.out.println(studentList);

            //Compare by first name and then last name
            Comparator<student> compareByRollName = Comparator.comparing(student::getRoll).reversed()
                .thenComparing(student::getName);

            Collections.sort(studentList, compareByRollName);

            System.out.println("Sorted based on reversed roll number and ascending name:");

            System.out.println(studentList);
        }
        catch (Exception ex)
        {
        }
    }
}
```

```
        ex.printStackTrace();
    }
}
```

▼ Summary

An interface declaration can contain method signatures, default methods, static methods and constant definitions. The only methods that have implementations are default and static methods.

A class that implements an interface must implement all the abstract methods declared in the interface.

An interface name can be used anywhere a type can be used.

▼ Inheritance

▼ What you can do in a subclass?

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- You can declare a **field** in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can write a new **static method** in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can write a new **instance method** in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.
- The inherited fields can be used directly, just like any other fields.
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can declare new methods in the subclass that are not in the superclass.

▼ Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a **public or protected nested class inherited** by a subclass has indirect access to all of the private members of the superclass.

▼ Casting

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then `obj` is **both** an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is *not* a `MountainBike`). This is called **implicit casting**.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can *tell* the compiler that we promise to assign a `MountainBike` to `obj` by **explicit casting**:

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `MountainBike` at runtime, an exception will be thrown.

Note:

You can make a logical test as to the type of a particular object using the ***instanceof*** operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {
    MountainBike myBike = (MountainBike)obj;
}
```

Here the ***instanceof*** operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

▼ Multiple inheritance of state, implementation, and type

▼ State

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. In addition, you can instantiate a class to create an object, which you cannot do with interfaces. As explained in the section [What Is an Object?](#), an object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of *multiple inheritance of state*, which is the ability to inherit fields from multiple classes.

For example, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor

will take precedence? **Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.**

▼ Implementation

Multiple inheritance of implementation is the ability to inherit method definitions from multiple classes.

Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they **sometimes cannot determine which member or method to access** or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass.

Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. **The Java compiler provides some rules to determine which default method a particular class uses.**

Note: Static methods in interfaces are never inherited.

▼ Type

The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any **class that implements the interface**. This is discussed in the section Using an Interface as a Type.

```
Shape rect1 = new Rectangle(corner1, corner2);
Rectangle rect2 = new Rectangle(corner1, corner2)
```

▼ Overriding and Hiding Methods

▼ Static method

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is `Animal`, which contains one instance method and one static method:

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```

public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        Animal.testClassMethod();
        myAnimal.testInstanceMethod();
    }
}

```

The `Cat` class overrides the instance method in `Animal` and hides the static method in `Animal`.

The `main` method in this class creates an instance of `Cat` and invokes `testClassMethod()` on the class and `testInstanceMethod()` on the instance.

The output from this program is as follows:

```

The static method in Animal
The instance method in Cat

```

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is

▼ Interface method

Default methods and abstract methods in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

- **Instance methods are preferred** over interface default methods.

Consider the following classes and interfaces:

```

public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}

```

```

public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}

```

```

public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}

```

```

public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}

```



```
}  
}
```

The method `Pegasus.identifyMyself` returns the string `I am a horse.`

- Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

Consider the following interfaces and classes:

```
public interface Animal {  
    default public String identifyMyself() {  
        return "I am an animal.";  
    }  
}
```

```
public interface EggLayer extends Animal {  
    default public String identifyMyself() {  
        return "I am able to lay eggs.";  
    }  
}
```

```
public interface FireBreather extends Animal { }
```

```
public class Dragon implements EggLayer, FireBreather {  
    public static void main (String... args) {  
        Dragon myApp = new Dragon();  
        System.out.println(myApp.identifyMyself());  
    }  
}
```

The method `Dragon.identifyMyself` returns the string `I am able to lay eggs.`

If two or more independently defined **default methods conflict**, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must **explicitly override the supertype methods**.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (`OperateCar` and `FlyCar`) that provide default implementations for the same method, (`startEngine`):

```
public interface OperateCar {  
    // ...  
    default public int startEngine(EncryptedKey key) {  
        // Implementation  
    }  
}
```

```
public interface FlyCar {  
    // ...  
    default public int startEngine(EncryptedKey key) {  
        // Implementation  
    }  
}
```

A class that implements both `OperateCar` and `FlyCar` must override the method `startEngine`. You could invoke any of the of the default implementations with the `super` keyword.

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

The name preceding `super` (in this example, `FlyCar` or `OperateCar`) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the `super` keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {
    String identifyMyself();
}
```

```
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
```

```
public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself` returns the string `I am a horse`. The class `Mustang` inherits the method `identifyMyself` from the class `Horse`, **which overrides the abstract method of the same name in the interface `Mammal`**.

▼ Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

▼ Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

	Superclass Instance Method	Superclass Static Method
--	----------------------------	--------------------------

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Note:

In a subclass, you can **overload** the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

▼ Polymorphism

Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as *virtual method invocation* and demonstrates an aspect of the important polymorphism features in the Java language.

```
// Java Program for Method Overriding

// Class 1
// Helper class
class Parent {

    // Method of parent class
    void Print()
    {
        // Print statement
        System.out.println("parent class");
    }
}

// Class 2
// Helper class
class subclass1 extends Parent {
    void Print() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
class subclass2 extends Parent {
    void Print() { System.out.println("subclass2"); }
}

// Class 4
// Main class
class trial {
    public static void main(String[] args)
    {
        // Creating object of class 1
        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Difference between

1. Child p = new Child()
2. Parent p = new Child()

1 Simply means, you're creating a new object of the datatype "Child"

2 Means you're creating a new object of the datatype "parent"

Where:

p is the variable name,

Parent and Child are datatypes

new is creating a new object

Child() is the constructor

When to use which one?

C c= new C();	P p=new C()
1. If we know exact runtime type of object then we should use this approach	1. if we don't know exact runtime type of object then we should use this approach. (polymorphism)
2. By using child reference we can call both parent and child class methods.	2. By using parent reference we can call only methods available in parent class and child specific methods we cant call.
3. we can use child reference to hold only for that particular child class object only.	3. we can use parent reference to hold any child class object.

Eg. ArrayList/Vector/Stack/LinkedList C(){}

So, we don't know what it returns hence, **List p = new C()** would be able to capture any of these class. But we can't do p.pop() as it is child specific method.

▼ Hiding fields

Within a class, a field that has the same name as a field in the superclass **hides the superclass's field, even if their types are different**. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through `super`, which is covered in the next section.

▼ Using Super keyword

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`. You can also use `super` to refer to a hidden field (although hiding fields is discouraged).

Consider this class, `Superclass`:

```
public class Superclass {  
    public void printMethod() {
```

```

        System.out.println("Printed in Superclass.");
    }
}

```

Here is a subclass, called `Subclass`, that overrides `printMethod()`:

```

public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }

    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}

```

Within `Subclass`, the simple name `printMethod()` refers to the one declared in `Subclass`, which overrides the one in `Superclass`. So, to refer to `printMethod()` inherited from `Superclass`, `Subclass` must use a qualified name, using `super` as shown. Compiling and executing `Subclass` prints the following:

```

Printed in Superclass.
Printed in Subclass

```

Subclass Constructors

The following example illustrates how to use the `super` keyword to invoke a superclass's constructor. Recall from the `Bicycle` example that `MountainBike` is a subclass of `Bicycle`. Here is the `MountainBike` (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```

public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                    int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}

```

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```
super();
```

or:

```
super(parameter list);
```

With `super()`, the superclass no-argument constructor is called. With `super(parameter list)`, the superclass constructor with a matching parameter list is called.

Note:

If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error.

`Object` *does* have such a constructor, so if `Object` is the only superclass, there is no problem.

▼ Final class and methods

You can declare some or all of a class's methods *final*. You use the `final` keyword in a method declaration to indicate that the method **cannot be overridden by subclasses**. The `Object` class does this—a number of its methods are `final`.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the `getFirstPlayer` method in this `ChessAlgorithm` class final:

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

- **Methods called from constructors should generally be declared final.** If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.
- Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when **creating an immutable class** like the `String` class.

▼ Abstract methods and classes

▼ Introduction

An *abstract class* is a class that is declared `abstract`—it **may or may not include abstract methods**. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself *must* be declared `abstract`, as in:

```
public abstract class GraphicObject {
    // declare fields
    // declare nonabstract methods
    abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared `abstract`.

Note: Methods in an *interface* (see the [Interfaces](#) section) that are not declared as default or static are *implicitly* abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

▼ Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. **However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.** With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public.

In addition, you can **extend only one class**, whether or not it is abstract, whereas **you can implement any number of interfaces**.

Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
 - You want to share code among several closely related classes.
 - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as **protected** and **private**).
 - You want to declare **non-static or non-final fields**. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
 - You expect that **unrelated classes** would implement your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
 - You want to specify the behavior of a particular data type, but not concerned about **who implements its behavior**.
 - You want to take advantage of **multiple inheritance** of type.

An example of an abstract class in the JDK is `AbstractMap`, which is part of the Collections Framework. Its subclasses (which include `HashMap`, `TreeMap`, and `ConcurrentHashMap`) share many methods (including `get`, `put`, `isEmpty`, `containsKey`, and `containsValue`) that `AbstractMap` defines.

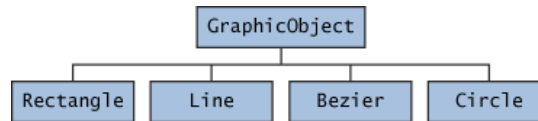
An example of a class in the JDK that implements several interfaces is `HashMap`, which implements the interfaces `Serializable`, `Cloneable`, and `Map<K, V>`. By reading this list of interfaces, you can infer that an instance of `HashMap` (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section [Serializable Objects](#)), and has the functionality of a map. In addition, the `Map<K, V>` interface has been enhanced with many default methods such as `merge` and `forEach` that older classes that have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the `HashMap` class implements several interfaces and also extends the abstract class `AbstractMap`.

▼ An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: `moveTo`, `rotate`, `resize`, `draw`) in common. Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and `moveTo`). Others require different implementations (for example, `resize` or `draw`). All `GraphicObject`s must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, `GraphicObject`) as shown in the following figure.

Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject



First, you declare an abstract class, `GraphicObject`, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the `moveTo` method. `GraphicObject` also declares abstract methods for methods, such as `draw` or `resize`, that need to be implemented by all subclasses but must be implemented in different ways. The `GraphicObject` class can look something like this:

```

abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
  
```

Each nonabstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw` and `resize` methods:

```

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
  
```

▼ When an Abstract Class Implements an Interface

In the section on [Interfaces](#), it was noted that a class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be `abstract`. For example,

```

abstract class X implements Y {
    // implements all but one method of Y
}

class XX extends X {
    // implements the remaining method in Y
}
  
```

In this case, class `X` must be `abstract` because it does not fully implement `Y`, but class `XX` does, in fact, implement `Y`.

▼ Class Members

An abstract class may have `static` fields and `static` methods. You can use these static members with a class reference (for example, `AbstractClass.staticMethod()`) as you would with any other class.

▼ Object as superclass

The `Object` class, in the `java.lang` package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the `Object` class. Every class you use or write inherits the instance methods of `Object`. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from `Object` that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException` Creates and returns a copy of this object.
- `public boolean equals(Object obj)` Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- `public final Class getClass()` Returns the runtime class of an object.
- `public int hashCode()` Returns a hash code value for the object.
- `public String toString()` Returns a string representation of the object.

The `notify`, `notifyAll`, and `wait` methods of `Object` all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

Note: There are some subtle aspects to a number of these methods, especially the clone method.

The clone() Method

If a class, or one of its superclasses, implements the `Cloneable` interface, you can use the `clone()` method to create a copy from an existing object. To create a clone, you write:

```
aCloneableObject .clone();
```

`Object`'s implementation of this method checks to see whether the object on which `clone()` was invoked implements the `Cloneable` interface. If the object does not, the method throws

a `CloneNotSupportedException` exception. Exception handling will be covered in a later lesson. For the moment, you need to know that `clone()` must be declared as

```
protected Object clone() throws CloneNotSupportedException
```

or:

```
public Object clone() throws CloneNotSupportedException
```

if you are going to write a `clone()` method to override the one in `Object`.

If the object on which `clone()` was invoked does implement the `Cloneable` interface, `Object`'s implementation of the `clone()` method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add `implements Cloneable` to your class's declaration. then your objects can invoke the `clone()` method.

For some classes, the default behavior of `Object`'s `clone()` method works just fine. If, however, an object contains a reference to an external object, say `ObjExternal`, you may need to override `clone()` to get correct behavior. Otherwise, a change in `ObjExternal` made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override `clone()` so that it clones the object *and* `ObjExternal`. Then the original object references `ObjExternal` and the clone references a clone of `ObjExternal`, so that the object and its clone are truly independent.

The equals() Method

The `equals()` method compares two objects for equality and returns `true` if they are equal. The `equals()` method provided in the `Object` class uses the identity operator (`==`) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The `equals()` method provided by `Object` tests whether the object *references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the `equals()` method. Here is an example of a `Book` class that overrides `equals()`:

```
public class Book {
    String ISBN;

    public String getISBN() {
        return ISBN;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

Consider this code that tests two instances of the `Book` class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays `objects are equal` even though `firstBook` and `secondBook` reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the `equals()` method if the identity operator is not appropriate for your class.

Note:

If you override `equals()`, you must override `hashCode()` as well.

The finalize() Method

The `Object` class provides a callback method, `finalize()`, that *may be* invoked on an object when it becomes garbage. `Object`'s implementation of `finalize()` does nothing—you can override `finalize()` to do cleanup, such as freeing resources.

The `finalize()` method *may be* called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect `finalize()` to close them for you, you may run out of file descriptors.

```
// Java code to show the
// overriding of finalize() method
import java.lang.*;

// Defining a class demo since every java class
// is a subclass of predefined Object class
// Therefore demo is a subclass of Object class
public class demo {
    protected void finalize() throws Throwable
    {
        try {

            System.out.println("inside demo's finalize()");
        }
        catch (Throwable e) {

            throw e;
        }
        finally {

            System.out.println("Calling finalize method"
                               + " of the Object class");

            // Calling finalize() of Object class
            super.finalize();
        }
    }

    // Driver code
    public static void main(String[] args) throws Throwable
    {

        // Creating demo's object
        demo d = new demo();

        // Calling finalize of demo
        d.finalize();
    }
}
```

The getClass() Method

You cannot override `getClass`.

The `getClass()` method returns a `Class` object, which has methods you can use to get information about the class, such as its name (`getSimpleName()`), its superclass (`getSuperclass()`), and the interfaces it implements (`getInterfaces()`). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
        obj.getClass().getSimpleName());
}
```

The `Class` class, in the `java.lang` package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (`isAnnotation()`), an interface (`isInterface()`), or an enumeration (`isEnum()`). You can see what the object's fields are (`getFields()`) or what its methods are (`getMethods()`), and so on.

The `hashCode()` Method

The value returned by `hashCode()` is the object's hash code, which is an integer value generated by a hashing algorithm.

By definition, if two objects are equal, their hash code *must also* be equal. If you override the `equals()` method, you change the way two objects are equated and `Object`'s implementation of `hashCode()` is no longer valid. Therefore, if you override the `equals()` method, you must also override the `hashCode()` method as well.

The `toString()` Method

You should always consider overriding the `toString()` method in your classes.

The `Object`'s `toString()` method returns a `String` representation of the object, which is very useful for debugging. The `String` representation for an object depends entirely on the object, which is why you need to override `toString()` in your classes.

You can use `toString()` along with `System.out.println()` to display a text representation of an object, such as an instance of `Book`:

```
System.out.println(firstBook.toString());
```

which would, for a properly overridden `toString()` method, print something useful, like this:

```
ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition
```

▼ Summary

Except for the `Object` class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide fields or methods that it inherits. (Note that hiding fields is generally bad programming practice.)

The table in [Overriding and Hiding Methods](#) section shows the effect of declaring a method with the same signature as a method in the superclass.

The `Object` class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from `Object` include `toString()`, `equals()`, `clone()`, and `getClass()`.

You can prevent a class from being subclassed by using the `final` keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.

An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.

▼ Number

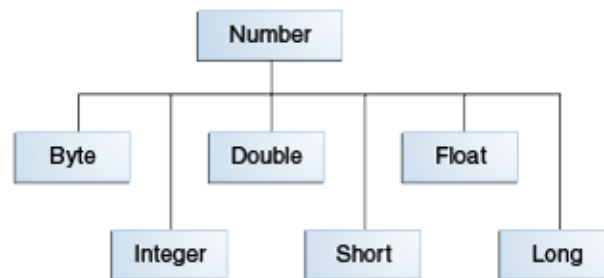
▼ The number classes

There are, however, reasons to use objects in place of primitives, and the Java platform provides *wrapper* classes for each of the primitive data types. These classes "wrap" the primitive in an object.

Often, the wrapping is done by the compiler—if you use a primitive where an object is expected, the compiler *boxes* the primitive in its wrapper class for you. Similarly, if you use a number object when a primitive is

expected, the compiler **unboxes** the object for you.

All of the numeric wrapper classes are subclasses of the abstract class `Number`:



Note:

There are four other subclasses of `Number` that are not discussed here. **`BigDecimal`** and **`BigInteger`** are used for **high-precision calculations**.

`AtomicInteger` and **`AtomicLong`** are used for **multi-threaded** applications.

There are three reasons that you might use a `Number` object rather than a primitive:

1. As an argument of a method that expects an object (often used when manipulating collections of numbers).
2. To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type. Eg. can be used in case you want to find maximum number using a loop and you should initialise the variable with `MIN_VALUE` to store the max value.
3. To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

The following table lists the instance methods that all the subclasses of the `Number` class implement.

Method	Description
<code>byte byteValue() short shortValue() int intValue() long longValue() float floatValue() double doubleValue()</code>	Converts the value of this <code>Number</code> object to the primitive data type returned.
<code>int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat) int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort)</code>	Compares this <code>Number</code> object to the argument.
<code>boolean equals(Object obj)</code>	Determines whether this number object is equal to the argument. The methods return <code>true</code> if the argument is not <code>null</code> and is an object of the same type and with the same numeric value. There are some extra requirements for <code>Double</code> and <code>Float</code> objects that are described in the Java API documentation.

Each `Number` class contains other methods that are useful for converting numbers to and from strings and for converting between number systems. The following table lists these methods in the `Integer` class. Methods for the other `Number` subclasses are similar:

Method	Description
<code>static Integer decode(String s)</code>	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
<code>static int parseInt(String s)</code>	Returns an integer (decimal only).

Method	Description
<code>static int parseInt(String s, int radix)</code>	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (<code>radix</code> equals 10, 2, 8, or 16 respectively) numbers as input.
<code>String toString()</code>	Returns a <code>String</code> object representing the value of this <code>Integer</code> .
<code>static String toString(int i)</code>	Returns a <code>String</code> object representing the specified integer.
<code>static Integer valueOf(int i)</code>	Returns an <code>Integer</code> object holding the value of the specified primitive.
<code>static Integer valueOf(String s)</code>	Returns an <code>Integer</code> object holding the value of the specified string representation.
<code>static Integer valueOf(String s, int radix)</code>	Returns an <code>Integer</code> object holding the integer value of the specified string representation, parsed with the value of <code>radix</code> . For example, if <code>s = "333"</code> and <code>radix = 8</code> , the method returns the base-ten integer equivalent of the octal number 333.

▼ Formatting numeric print output

▼ The `printf` and `format` methods

The `java.io` package includes a `PrintStream` class that has two formatting methods that you can use to replace `print` and `println`. These methods, `format` and `printf`, are equivalent to one another. The familiar `System.out` that you have been using happens to be a `PrintStream` object, so you can invoke `PrintStream` methods on `System.out`. Thus, you can use `format` or `printf` anywhere in your code where you have previously been using `print` or `println`. For example,

Converter	Flag	Explanation
d		A decimal integer.
f		A float.
n		A new line character appropriate to the platform running the application. You should always use <code>%n</code> , rather than <code>\n</code> .
tB		A date & time conversion—locale-specific full name of month.
td, te		A date & time conversion—2-digit day of month. <code>td</code> has leading zeroes as needed, <code>te</code> does not.
ty, tY		A date & time conversion— <code>ty</code> = 2-digit year, <code>tY</code> = 4-digit year.
tl		A date & time conversion—hour in 12-hour clock.
tM		A date & time conversion—minutes in 2 digits, with leading zeroes as necessary.
tp		A date & time conversion—locale-specific am/pm (lower case).
tm		A date & time conversion—months in 2 digits, with leading zeroes as necessary.
tD		A date & time conversion—date as <code>%tm%td%ty</code>
	08	Eight characters in width, with leading zeroes as necessary.
	+	Includes sign, whether positive or negative.
	,	Includes locale-specific grouping characters.
	-	Left-justified..
	.3	Three places after decimal point.
	10.3	Ten characters in width, right justified, with three places after decimal point.

The following program shows some of the formatting that you can do with `format`. The output is shown within double quotes in the embedded comment:

```

import java.util.Calendar;
import java.util.Locale;

public class TestFormat {

    public static void main(String[] args) {
        long n = 461012;
        System.out.format("%d\n", n);        // --> "461012"
        System.out.format("%08d\n", n);      // --> "00461012"
        System.out.format("%+8d\n", n);      // --> " +461012"
        System.out.format("% ,8d\n", n);     // --> " 461,012"
        System.out.format("%+,8d\n\n", n);   // --> "+461,012"

        double pi = Math.PI;

        System.out.format("%f\n", pi);        // --> "3.141593"
        System.out.format("%.3f\n", pi);      // --> "3.142"
        System.out.format("%10.3f\n", pi);    // --> "      3.142"
        System.out.format("%-10.3f\n", pi);   // --> "3.142"
        System.out.format(Locale.FRANCE,
            "%-10.4f\n\n", pi); // --> "3,1416"

        Calendar c = Calendar.getInstance();
        System.out.format("%tB %te, %tY\n", c, c, c); // --> "May 29, 2006"

        System.out.format("%tl:%tM %tp\n", c, c, c); // --> "2:34 am"

        System.out.format("%tD\n", c);        // --> "05/29/06"
    }
}

```

▼ The DecimalFormat Class

```

import java.text.*;

public class DecimalFormatDemo {

    static public void customFormat(String pattern, double value ) {
        DecimalFormat myFormatter = new DecimalFormat(pattern);
        String output = myFormatter.format(value);
        System.out.println(value + " " + pattern + " MAIN OUTPUT -> "
            + output);
    }

    static public void main(String[] args) {

        customFormat("###,###.###", 123456.789);
        customFormat("###.##", 123456.789);
        customFormat("000000.000", 123.78);
        customFormat("$###,###.###", 12345.67);
    }
}

```

The output is:

```

123456.789  ###,###.### MAIN OUTPUT -> 123,456.789
123456.789  ###.## MAIN OUTPUT -> 123456.79
123.78     000000.000 MAIN OUTPUT -> 000123.780
12345.67   $###,###.### MAIN OUTPUT -> $12,345.67


```

▼ Beyond basic maths

Math.cos(angle), Math.PI, Math.E, Math.floor(n), Math.max(a,b), exp, log, pow, sqrt

Beyond Basic Arithmetic

The Java programming language supports basic arithmetic with its arithmetic operators: +, -, *, /, and %. The class in the java.lang package provides methods and constants for doing more advanced mathematical computation.

 <https://docs.oracle.com/javase/tutorial/java/data/beyondmath.html>

▼ Character

Most of the time, if you are using a single character value, you will use the primitive `char` type. For example:

```
char ch = 'a';
// Unicode for uppercase Greek omega character
char uniChar = '\u03A9';
// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

There are times, however, when you need to use a char as an object—for example, as a method argument where an object is expected. The Java programming language provides a *wrapper* class that “wraps” the `char` in a `Character` object for this purpose. An object of type `Character` contains a single field, whose type is `char`. This `Character` class also offers a number of useful class (that is, static) methods for manipulating characters.

You can create a `Character` object with the `Character` constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a `Character` object for you under some circumstances. For example, if you pass a primitive `char` into a method that expects an object, the compiler automatically converts the `char` to a `Character` for you. This feature is called *autoboxing*—or *unboxing*, if the conversion goes the other way. For more information on autoboxing and unboxing, see [Autoboxing and Unboxing](#).

Note: The `Character` class is immutable, so that once it is created, a `Character` object cannot be changed.

The following table lists some of the most useful methods in the `Character` class, but is not exhaustive. For a complete listing of all methods in this class (there are more than 50), refer to the [java.lang.Character](#) API specification.

Method	Description
<code>boolean isLetter(char ch) boolean isDigit(char ch)</code>	Determines whether the specified char value is a letter or a digit, respectively.
<code>boolean isWhitespace(char ch)</code>	Determines whether the specified char value is white space.
<code>boolean isUpperCase(char ch) boolean isLowerCase(char ch)</code>	Determines whether the specified char value is uppercase or lowercase, respectively.
<code>char toUpperCase(char ch) char toLowerCase(char ch)</code>	Returns the uppercase or lowercase form of the specified char value.
<code>toString(char ch)</code>	Returns a <code>String</code> object representing the specified character value — that is, a one-character string.

Escape Sequences

A character preceded by a backslash (\) is an *escape sequence* and has special meaning to the compiler. The following table shows the Java escape sequences:

Escape Sequence	Description
<code>\t</code>	Insert a tab in the text at this point.
<code>\b</code>	Insert a backspace in the text at this point.
<code>\n</code>	Insert a newline in the text at this point.

Escape Sequence	Description
<code>\r</code>	Insert a carriage return in the text at this point.
<code>\f</code>	Insert a form feed in the text at this point.
<code>\'</code>	Insert a single quote character in the text at this point.
<code>\"</code>	Insert a double quote character in the text at this point.
<code>\\</code>	Insert a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly. For example, if you want to put quotes within quotes you must use the escape sequence, `\`, on the interior quotes. To print the sentence

```
She said "Hello!" to me.
```

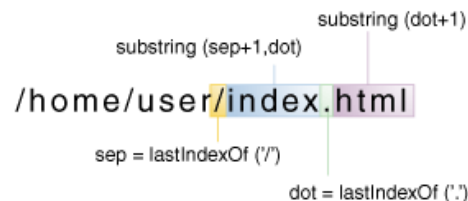
you would write

```
System.out.println("She said \"Hello!\" to me.");
```

▼ String

```
String string1 = "Happy";
System.out.println(string1.substring(0,2));    //Output = Ha
String string1 = "Hello World";
int index = string1.indexOf("World");    //Output = 6. -1 if not found

//Finding occurrences of substring:
String theString = "is this good/is?";
String substring = "is";
int index = theString.indexOf(substring);
while(index != -1) {
    System.out.print(index + " ");
    index = theString.indexOf(substring, index + 1);    //Output = 0 5 13
}
```



▼ StringBuilder class

`StringBuilder` objects are like `String` objects, except that they can be modified. Internally, these objects are treated like variable-length arrays that contain a sequence of characters. At any point, the length and content of the sequence can be changed through method invocations.

StringBuffer = mutable sequence of characters. **Guarantee** of synchronization

StringBuilder = mutable sequence of characters. **No guarantee** of synchronization

<https://www.youtube.com/watch?v=oYcb0N1YfVw&list=PLsyebzWxl7oZ-fxDYkOTOURHhMuWD1BK&index=105>

▼ Autoboxing and unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on.

If the conversion goes the other way, this is called **unboxing**.

Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the [Generics \(Updated\)](#) lesson.

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because **li is a list of Integer objects, not a list of int values**, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an Integer object from i and adds the object to li. Thus, the compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
    return sum;
}
```

Because the **remainder (%) and unary plus (+=) operators do not** apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
    return sum;
}
```

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.

The `Unboxing` example shows how this works:

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // π is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

▼ Generics

▼ Introduction

▼ Video link

Generics in java:

<https://www.youtube.com/watch?v=XMvznsY02Mk&list=PLsyeobzWxl7oZ-fxDYkOToURHhMuWD1BK&index=134>

Some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time.

Why use generics?

In a nutshell, generics **enable types** (classes and interfaces) **to be parameters when defining classes**, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time. A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- Elimination of casts. The following code snippet without generics requires casting: When re-written to use generics, the code does not require casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);    // no cast
```

- Enabling programmers to implement generic algorithms. By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Invoking and Instantiating a Generic Type

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — Integer in this case — to the Box class itself.

```
Box<Integer> integerBox;
```

An invocation of a generic type is generally known as a *parameterized type*.

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

This pair of angle brackets, <>, is informally called *the diamond*.

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

▼ An example

A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any **interface** type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

▼ Type parameter naming conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key

- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

▼ Multiple Type parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to [autoboxing](#), it is valid to pass a `String` and an `int` to the class.

As mentioned in [The Diamond](#), because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (that is, `K` or `V`) with a parameterized type (that is, `List<String>`). For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

▼ Raw types

NOT USED NOW

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {  
    public void set(T t) { /* ... */ }  
    // ...  
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

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

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, **a non-generic class or interface type is *not* a raw type.**

Raw types show up in legacy code because lots of API classes (such as the `Collections` classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you `Objects`. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();  
Box rawBox = stringBox; // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box(); // rawBox is a raw type of Box<T>  
Box<Integer> intBox = rawBox; // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();  
Box rawBox = stringBox;  
rawBox.set(8); // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The [Type Erasure](#) section has more information on how the Java compiler uses raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {  
    public static void main(String[] args){  
        Box<Integer> bi;  
        bi = createBox();  
    }  
  
    static Box createBox(){  
        return new Box();  
    }  
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
    bi = createBox();
           ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag.

The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see [Annotations](#).

▼ Generic methods

Generic methods are methods that introduce their own type parameters. This is **similar to declaring a generic type, but the type parameter's scope is limited to the method** where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the **type parameter section must appear before the method's return type**.

The Util class includes a generic method, `compare`, which compares two Pair objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
            p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed.

This feature, known as **type inference**, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, [Type Inference](#).

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```


▼ Bounded type parameters

See bounded type parameters program.

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, followed by the `extends` keyword, followed by its *upper bound*, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
public class Box<T> {  
    private T t;  
  
    public void set(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
  
    public <U extends Number> void inspect(U u){  
        System.out.println("T: " + t.getClass().getName());  
        System.out.println("U: " + u.getClass().getName());  
    }  
  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>();  
        integerBox.set(new Integer(10));  
        integerBox.inspect("some text"); // error: this is still String!  
    }  
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of `inspect` still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot  
    be applied to (java.lang.String)  
        integerBox.inspect("10");  
                        ^  
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {  
    private T n;  
  
    public NaturalNumber(T n) { this.n = n; }  
  
    public boolean isEven() {  
        return n.intValue() % 2 == 0;  
    }  
  
    // ...  
}
```

The `isEven` method invokes the `intValue` method defined in the `Integer` class through `n`.

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have *multiple bounds*:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. **If one of the bounds is a class, it must be specified first.** For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

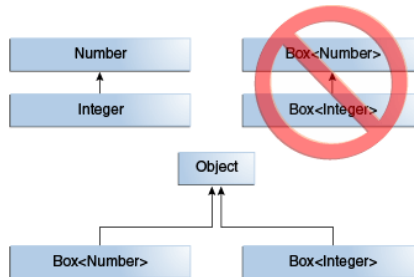
```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

▼ Generics, inheritance and subtypes

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.

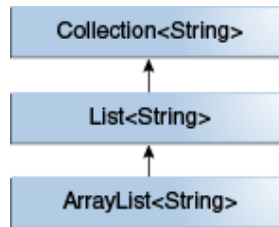
For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see [Wildcards and Subtyping](#).

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the `extends` and `implements` clauses.

Using the Collections classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

A sample Collections hierarchy



▼ Type inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the *most specific* type that works with all of the arguments.

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets.

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (`<>`) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

▼ Wildcards

In generic code, the question mark (`?`), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

- Upper Bounded Wildcards
 - You keep the code open for extension to support any new types getting added to the type hierarchy.
- Lower Bounded Wildcards
 - You close the code to support any new types in the hierarchy and restrict to the types currently present .

<https://www.youtube.com/watch?v=1Omf1Vba44Y>

▼ Code

```
package com.java.languagebasics;

// Glass
// Liquid <- Juice <- OrangeJuice <- Mirinda

class Glass<T>{}

interface Liquid{}

class Juice implements Liquid{}

class OrangeJuice extends Juice{}
```

```

class Mirinda extends OrangeJuice{}

class Tray
{
    void add(Glass<? extends Juice> g1){}
    //this is upper bounded. Provides bound at upper end. Liquid <- JUICE <- ORANGEJUICE <- MIRINDA

    void remove(Glass<? super Juice> g2){}
    //this is lower bounded. Provides bound at lower end. LIQUID <- JUICE <- OrangeJuice <- Mirinda
}

public class BoundedWildcards
{
    public static void main(String[] args)
    {
        try
        {
            Tray t2 = new Tray();

            t2.add(new Glass<>()); //type inference would be acting over here

            t2.add(new Glass<Juice>());

            t2.add(new Glass<OrangeJuice>());

            // t2.add(new Glass<Liquid>()); //Liquid will give error. As liquid does not extend juice

            t2.remove(new Glass<Juice>());

            t2.remove(new Glass<Liquid>());

            // t2.remove(new Glass<Mirinda>()); //error as it crosses the lower bound

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}

```

▼ Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as **wildcard capture**.

```

import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}

```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a **private helper method** which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper`, as shown in [WildcardFixed](#):

```

public class WildcardFixed {

```

```

void foo(List<?> i) {
    fooHelper(i);
}

// Helper method created so that the wildcard can be captured
// through type inference.
private <T> void fooHelper(List<T> l) {
    l.set(0, l.get(0));
}
}

```

▼ Type erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or 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 parameterized types; consequently, generics incur no runtime overhead.

<https://www.youtube.com/watch?v=6HoPGat-rqM>

▼ Restrictions on generic

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```

class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

```

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

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```
Pair< int, char > p = new Pair<>(8, 'a'); // compile-time error
```

You can substitute only non-primitive types for the type parameters K and V:

```
Pair< Integer, Character > p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

For more information on autoboxing, see [Autoboxing and Unboxing](#) in the [Numbers and Strings](#) lesson.

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E(); // compile-time error
    list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance(); // OK
    list.add(elem);
}
```

You can invoke the append method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. **You cannot, therefore, create static fields of type parameters.**

Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) { // compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the `rtti` method is:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`. The most you can do is to use an unbounded wildcard to verify that the list is an `ArrayList`:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) { // OK; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number> ln = (List<Number>) li; // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1; // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi"; // OK
strings[1] = 100; // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[2]; // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>(); // OK
stringLists[1] = new ArrayList<Integer>(); // An ArrayStoreException should be thrown,
// but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ }    // compile-time error

// Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ } // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {    // compile-time error
        // ...
    }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {    // OK
        // ...
    }
}
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

▼ Package

To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so *types* are often referred to in this lesson simply as *classes and interfaces*.