Tuesday, 7 March 2017

Learning Java: Core Java Volume 1

Ankit Kumar

- **An Introduction to Java**

- Java is a whole *platform*, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection. Java has everything—a good language, a high-quality execution environment, and a vast library.

- **Java was designed to be:**

  • **Simple**

    - We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible. Java omits many rarely used, poorly understood, confusing features of C++ that, in our experience, bring more grief than benefit

    - Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone on small machines. The size of the basic interpreter and class support is about 40K; the basic standard libraries and thread support (essentially a self-contained microkernel) add another 175K.

  • **Object-Oriented**

    - Simply stated, object-oriented design is a programming technique that focuses on the data (= objects) and on the interfaces to that object. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he is building, and secondarily with the tools used to make it; a "non-object-oriented" carpenter would think primarily of his tools. The object-oriented facilities of Java are essentially those of C++.

    - The major difference between Java and C++ lies in multiple inheritance, which Java has re- placed with the simpler concept of interfaces.

  • **Distributed**

    - Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.

  • **Robust**

    - Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error-prone. . . The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

- **Secure**
  - Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.
  - From the beginning, Java was designed to make certain kinds of attacks impossible, among them:
    - Overrunning the runtime stack—a common attack of worms and viruses
    - Corrupting memory outside its own process space
    - Reading or writing files without permission
- **Architecture-Neutral**
  - The compiler generates an architecture-neutral object file format—the compiled code is executable on many processors, given the presence of the Java runtime system. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.
  - However, virtual machines have the option of translating the most frequently executed bytecode sequences into machine code—a process called just-in-time compilation. Java's virtual machine has another advantage. It increases security because it can check the behaviour of instruction sequences.
- **Portable**
  - Unlike C and C++, there are no "implementation-dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of arithmetic on them.
  - Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. Strings are saved in a standard Unicode format.
  - You can work with files, regular expressions, XML, dates and times, databases, network connections, threads, and so on, without worrying about the underlying operating system.
- **Interpreted**
  - The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.
- **High-Performance**
  - While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on.
- **Multithreaded**

- The benefits of multithreading are better interactive responsiveness and real-time behaviour. Nowadays, we care about concurrency because Moore's law is coming to an end. Instead of faster processors, we just get more of them, and we have to keep them busy.

The Java Programming Environment

Using JDK

- Java Development Kit (JDK) includes both the compiler as well as the interpreter for development purposes. Java Runtime Environment (JRE) contains the virtual machine but not the compiler.
  - When you install the JDK on Linux, you need to carry out one additional step: Add the `jdk/bin` directory to the executable path—the list of directories that the operating system traverses to locate executable files.
  - On Linux, add a line such as the following to the end of your `~/.bashrc` or `~/.bash_profile` file:

```
export PATH=jdk/bin:$PATH
```

  - The `javac` program is the Java compiler. It compiles the file, for example, `Welcome.java` into the file `Welcome.class`. The `java` program launches the Java virtual machine. It executes the bytecodes that the compiler placed in the class file.

Fundamental Programming Structures in Java

- **Read from Book Core Java Vol 1, Page 41-129**

Objects and Classes

Introduction to Object Oriented Programming

- An object oriented program is made of objects. Each object has a specific functionality, exposed to its users, and a hidden implementation.
- Traditional structured programming consists of designing a set of procedures or algorithms to solve a problem. Once the procedures are determined, the traditional next step was to find appropriate ways to store data.

- OOP reverses the order: puts the data first, then looks at the algorithms to operate on the data.
- **Classes**
  - A Class is the template or blueprint from which objects are made. When you construct an object from a class, you are said to have created an instance of the class.
  - All code that you write in Java goes inside classes.
  - Java libraries supply thousands of classes. But still you have to create your own classes to describe the objects of your application's problem domain.
  - **Encapsulation** is a key concept in working with objects. It consists of combining data and behaviour in one package and hiding the implementation details from the users of the object.
  - The bits of data in an object are called **instance fields**, and the procedures that operate on the data are called its **methods**.
  - A specific object that is an instance of a class will have specific values of its instance fields. The set of those values defines the current **state** of the object. Whenever you invoke a method on an object, its state may change.
  - The key to making encapsulation work is to have methods never directly access instance fields in a class other than their own. Programs should interact with object data only through the object's methods.
  - Encapsulation is a way to give an object its black-box behaviour which is the key to reuse and reliability.
  - Classes can be built by extending other classes. Java, in fact, comes with a class **Object** which is extended by every other class in Java. All classes in Java, by default, extend the Object class.
  - When you extend an existing class, the new class has all the properties and methods the class that you extend. You then supply new methods and data fields that apply to your new more specific class only.
  - The concept of extending a class to obtain another class is called inheritance.
- **Objects**
  - Objects have three key characteristics:
    - objects behaviour - defined by the methods you can call on it
    - objects state - set of value of the instance fields of the objects. A change in the objects state must be a consequence of method calls.

- objects identity - apart form objects of different classes even the objects that are instances of the same class differs in their identity and usually differ in their state.
- These key characteristics can influence each other.

- **Identifying Classes**
  - To begin designing an Object Oriented Program: Identify your classes and the add methods to each class.
  - A simple rule of thumb in identifying classes is to look for nouns in the problem statement/analysis. Methods, on the other hand, correspond to verbs.
  - With each verb you then identify the object that has the major responsibility for carrying it out and then add it to the class the object belongs to.

- **Relationship between Classes**
  - The most common relationship between classes are :
    - Dependence ("uses-a")
    - Aggregation ("has-a")
    - Inheritance ("is-a")
  - A class **depends** on another class if its methods use or manipulate objects of that class.
  - You should always try to minimise the number of classes that depend on each other. If a class A is unaware of the existence of a class B, it is also unconcerned about any changes to Band this means that changes to B do not introduce bugs into A. This is called minimising **coupling** between classes.
  - **Containment** means that objects of class A contain objects of class B.
  - The **inheritance** relationship expresses a relationship between a more special and a more general class.
  - In general, if class A extends class B, then class A inherits the methods from class B and has more capabilities than class B.
  - Many programmers use the UML (Unified Modelling Language) notation to draw class diagrams that describe the relationships between classes.

- **Using Predefined Classes**
  - **Objects and Object Variables**

- To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.
- In the Java programming language, you use **constructors** to construct new instances. A constructor is a special method whose purpose is to construct and initialise objects.
- **Constructors** always have the same name as the class name. To construct an object of a class, you combine the appropriate constructor of the class with the **new** operator.

```
new ClassName(…initial state constructor arguments…);
```

- The **new** operator calls the constructor of the class that is specified next to it and creates an object of that class on the heap. It then returns back a reference to that object on the heap.
- You can then pass the object reference to a method or invoke a method on that object reference or assign it to an object variable.
- There is an important difference between objects and object variables. Objects live on the heap. But when we declare an object variable, that object variable is not an object. Neither does it even refer to an object until we initialise it with an object reference.
- This means that you must always initialise the declared variables before you can use them. You can initialise an object variable with a reference to a newly constructed or you can set the variable to refer to an existing object.
- An object variable doesn't actually contain an object. It only refers to an object which is present on the **garbage collectible heap**.
- In Java, the value of an object variable is a reference to an object that is stored on the garbage collectible heap. As mentioned the return value of a new operator is an object reference.
- You can explicitly set an object variable **null** to indicate that it currently refers to no object. But if you apply a method to a variable that holds `null`, a runtime `NullPointerException` occurs. Calling a method on an object reference without initialising with any value also raises a compile time Exception.
- Local variables are not initialised automatically to `null`. You must initialise them either by calling new or initialising with another object reference variable of Covariant type or by setting them `null`.

- **LocalDate Class of the Java Library**
  - You do not use a constructor to construct objects of the **LocalDate** class. Instead you use static factory methods that call constructors on your behalf.

- `LocalDate.now()` constructs a new object that represents the date at which the object was constructed.
- You can construct an object of a specific date by supplying year, month and day. You can store reference to these objects in a LocalDate object variable.

```
LocalDate yesterday = LocalDate.of(2017,3,6);
```

- Once you have the a `LocalDate` object, you can find out the year, month and day with the methods `getYear`, `getMonthValue`, and `getDayOfMonth` like below:

```
int year = newYearsEve.getYear();
int month = newYearsEve.getMonthValue();
int day = newYearsEve.getDayOfMonth();
```

- Note: A method is deprecated when a library designer realises that the method should have never been introduced in the first place.

• **Mutator and Accessor Methods**

- Methods that only access objects without modifying them are called **accessor** methods. On the other hand, methods with mutate/change the state of an object are caller **mutator** methods.
- E.g. When you call `toUpperCase` on a string, that string stays the same and a new string with uppercase characters is returned.
- `java.time.LocalDate` class API

- **Defining Your Own Classes**

• To build a complete program, you combine several classes, one of which has a `main` method.

• The name of the source file containing a `public` class must be the same as that of the `public` class with extension `.java` appended. You can have only one `public` class in a source file but any number of nonpublic classes with it.

• You then start the program by giving the byte code interpreter the name of the class that contains the `main` method of your program. The byte code interpreter starts running the code in the `main` method in the given class.

• **Use of Multiple Source Files**

- You can invoke the java compiler with a wildcard. Then all the source files matching the wildcard will be compiled into class files like:

```
java Employee*.java
```

- Or you can simply give the source file that contains the `main` method that will be used for the purpose of launching the entire Application.

- In the second case, when the Java compiler sees a class being used inside the class with the main that is specified while invoking the compiler, it will look for that class' compiled `.class` file. If it doesn't find that file, it automatically searches for it's `.java` source file and complies it. Moreover if the timestamp of the version of this class' `.java` file that it finds is newer that that of its existing `.class` file, then the java compiler will automatically recompile the file.
- The keyword **public** means that any method in any class can call the method.
- The **private** keyword means that the only methods that can access private fields of a class or call private methods of a class are the methods of that class itself.
- It is strongly recommended to make all the instance fields of your class `private`.
- The name of the constructor is the same as that of the class. A class can have multiple constructors taking in different sets of parameters (method signatures) to set the different initial state.
- A constructor can only be called in conjunction with with the `new` operator. You can't apply the constructor to an existing object to reset the instance fields.
- Just keep the following in mind:
  - A constructor has the same name as the class.
  - A class can have more than one constructor.
  - A constructor can take zero, one, or more parameters.
  - A constructor has no return value.
  - A constructor is always called with the new operator.

- **Implicit and Explicit Parameters**
  - Methods operate on objects and access their instance fields.
  - The first parameter that is passed to the method is the object on which the method is invoked. It is implicitly passed as keyword `this`. The implicit keyword does not appear in the method declaration.
  - The other parameters that are passed inside the parentheses after the method name are called explicit parameters.
  - The methods can access an instance field either by its `<variableName>` or by using `this.<variableName>`
  - In every method, the keyword `this` refers to the implicit parameter. Some programmers prefer to access instance variables as `this.<variableName>` because it clearly distinguishes between instance fields and local variables.
- **Benefits of Encapsulation**

- Methods which simply return the values of instance fields are called **getters** or field accessors. In case of object variable instance fields, getters return the reference to the actual object. This must be viewed cautiously. Be careful not to return references to *mutable objects*.
- Using **setters** or Mutator methods to set the value for a field allows us to perform error checking whereas code that simply assigns to a field may set a faulty value.
- If you need to return a reference to a mutable object, you should first clone it and then return a reference to the clone object. A **clone** is an exact copy of an object stored in a new location. As a rule of thumb always use clone whenever you need to return a copy of a mutable field.

• **Class-Based Access Privileges**

- It's common knowledge that a method can access the private data of the object on which it is invoked.
- *But a method can also access the private data of all the objects of it's class*.
- So the method below of Employee class is legal as it is accessing the `name` field of another Employee class object `other`.

```
public boolean equals(Employee other) {

      return name.equals(other.name);

}
```

• **Private Methods**

- When implementing a class we make all data fields private because public data fields are dangerous.
- While most methods are public, private methods are useful in certain situations. Sometimes you may want to increase modularity by breaking up the computation into separate helper methods. Typically these helper methods shouldn't be a part of the public interface. Such methods are best implemented as `private` (meaning can only be called by other methods of their own class).
- To make a method `private` you simply make the access modifier in the method definition `private`.
- By making a method private you are under no obligation to keep it available if you change your implementation.
- The point is that as long as the method is `private`, the designers of the class can assume that it is never used outside of class so they can simply drop it if need be.
- If a method is `public`, you cannot simply drop it because other code might rely on that.

• **Final Instance Fields**

- You can define an instance field as `final`. Bust such a field must be initialised when the object is constructed. You must guarantee that the field value has been set after the end of every constructor. Afterwards the field cannot be modified again.
- The final modifier is useful for fields whose type is primitive or an immutable class. (A class is immutable if none of its methods ever mutate its objects)
- The `final` keyword merely means that the object reference stored in the object variables will never again refer to a different object. But in case of the object being referred being mutable, the state of an object can still be changed/mutated. Hence for mutable classes the `final` modifier can be confusing.

- Static Fields and Methods

- **Static Fields**
  - If you define a field as `static`, then there is only one such field per class. In contrast, each object gets its own copy of all the instance fields. In other words, the `static` fields are shared among all the instances of a class.
  - Even if there is no instance if a class yet created, the static fields remain present and accessible. It belongs to the class.
- **Static Constants**
  - Static variables are quite rare. However, static constants are more common.
  - Static constants include : `Math.Pi`, `System.out` ; had they not been static we would have need to construct an object of their classes to access these fields.
  - It's never good idea to have `public` fields but incase of `public static final` fields (i.e constants) its fine.
- **Static Methods**
  - Static Methods are methods that do not operate on objects. They have no implicit parameter (i.e. `this` parameter). In a non static method `this` parameter refers to the implicit parameter. `Math.pow` is a static method.
  - Static methods can only access the static fields of a class i.e static methods cannot access instance fields of a class because they don't operate on objects.
  - To access a static method you supply the name of a class like `ClassName.methodName(parameterList)`.
  - Omitting the static keyword from the method declaration would have meant that we would need an object of class to which the method belongs to in order to be able to invoke the method.
  - It is legal to use an object to call a static method. But we recommend that you use class names and not objects to invoke static methods.

- Use static method sin two situations:
    - When a method doesn't need access to the object state because all needed parameters are supplied as explicit parameters.
    - When a method only needs to access the static fields of the class.
- **Factory Methods**
    - Factory methods are static methods that calls the constructors on behalf od the users to construct objects of a class.
    - There are two reasons for exposing static factory methods instead of direct constructors:
        - You cant give names to constructors. The constructor name is always the same as that of the class. But if it is possible to get different nature of instances from a class then it makes more sense to expose two differently named static factory methods which constructs those different nature instances. E.g `NumberFormat` class exposes `getCurrencyInstance()` and `getPercentInstance()` static factory methods which calls the constructors to return two different nature of instances.
        - When you use a constructor you cant vary the types of the constructed object that you return. But factory methods usually return covariant object types.
- **The `main` method**
    - You can call static methods without having any objects just like you can access any static fields without any objects.
    - The `main` method is a static method. It does not operate on objects.In fact when the program starts there aren't any objects yet. Its the main method that constructs the objects that the program needs.
    - **Tip**: Every class can have a `main` method. This is a handy trick for unit testing of classes. And when you want to test only a single class in isolation you pass that class to the java byte interpreter and mock external dependencies to test that class. doing so would run only the main method in the class that you want to test and hence pass to the java byte code interpreter.

- Method Parameters

- A method uses parameters. A method is passed arguments.
- The term ***call by value*** means that the method gets just the value that the caller provides. In contrast, ***call by reference*** means that the method gets the location of the variable that the caller provides. So the method can modify the value stored in a variable passed by reference but not in one passed by value.
- ***The java programming language always uses call by value***. That means that the method gets a copy of all the parameter values. In particular the method cannot modify the contents of any parameter variables passed to it.

- There are however two kinds of method parameters:
  - Primitive Types
  - Object References
- While it is impossible for the method to change the value of a primitive type parameter variable (the arguments that is passed), it is easily possible to implement methods that change the state of an object parameter. Since the object variable parameter gets a copy of the object variable argument both the variables refer to the same object. And in case when the object belongs to a mutable class it is possible for the method to change the state of the object variable argument that it has been passed.
- Just to clarify, the method doesn't change the object references that are stored in the argument variables that is passed on the method parameters. The method parameters are simply initialised with copies of the argument variable values which are references to and not the location of the argument variable referenced objects. So the methods can only change the state of the objects references to whom are passed but not make a reference arguments variable refer to another object.
- Summary:
  - A method cannot modify a parameter of a primitive type (that is, numbers or boolean values).
  - A method can change the state of an object parameter.
  - A method cannot make an object parameter refer to a new object.
- **Think here:** In C/C++ when one passes variable address/reference through '&' they give the address of the variable to the method and then the method can change the value that the address holds and hence can change the object that the outside variables are referencing to at that location.

- Object Construction

- **Overloading**
  - Some classes have more than one constructor. This capability is called overloading.
  - Overloading occurs if several methods have the same name but different method signature i.e order and type of parameters that they take. The compiler must sort out the which method to call. It picks the correct method by matching the parameter types in the headers of the various methods with the types of values passed in a method call. A compile time error occurs if the compiler cannot match the parameters either because there is no match at all or because there is not one that better than all others. This process of finding a match is called overloading resolution.
  - Java allows you to overload any method—not just constructor methods. Thus to completely describe a method, you need to specify it name together with its parameter types.This is called the signature of the method.

- The return type is not a part of the method signature. That is you cannot have two methods with same names and same order and types of parameters but different return types.

- **Default field Initialisation**
  - If you don't set a field explicitly in a constructor, it is automatically set to a default value: numbers to `0`, `boolean` values to `false`, and object variables to `null`. Although its poor practice to rely on defaults as it makes it harder for other people to understand your code if the fields are being initialised invisibly.
  - You must always explicitly initialise local variables in a method because they don't get initialised with default values. Class fields, as mentioned in the point above, do get initialised with default values.
- **The Constructor with No Arguments**
  - Many classes contain a constructor with no arguments that created an object whose state is set to an appropriate default value as initialised by the programmer.
  - If you write a class with no constructor whatsoever, then a no-argument constructor is provided by the compiler. This constructor sets all the instance fields to their default values.
  - If a class supplies atleast one constructor but does not supply a no-argument constructor, then in this case the compiler **does not** provide a no-argument constructor and hence it becomes illegal to construct objects without supplying arguments.
  - Putting it again, you get a free no-argument constructor only when your class has no other constructors. It you write a class with even a single constructor of your own and you want the users of your class to have the ability to create an instance by calling the no-arguments constructor then you must explicitly provide the no-argument constructor. If you are happy with the default values for all fields, you can simply supply:

```
public ClassName(){

}
```

- **Explicit Field Initialisation**
  - By overloading the constructor methods in a class, you can build many ways to set the initial state of the instance fields your classes. But still, It is always a good idea to make sure that, regardless of the constructor call, every instance field is set to something meaningful. Like initialising a `String` with an empty string(“”) instead of letting it be `null`.
  - This assignment is carried out before the constructor executes. This syntax is particularly important if all the constructors of a class need to set a particular instance field to the same value. In that case you can simply factor out the common initialisation values and the initialise them while declaring them in the class definition.

- The initialisation value doesn't have to be a constant value. E.g

```
class Employee{

      //non-default meaningful value
      private String name = “”;
      private static int nextID;
      private int id = assignId();

      ……

      private static int assignID(){
            int r = nextId;
            nextId++;
            return r;
      }
}
```

- **Parameter Names**
  - A commonly used constructor or setter parameter naming pattern relies on the fact that the parameter variables shadow the instance fields with same name. Hence the instance fields are referred to as `this.variableName` and the parameter as `variableName`.

- **Calling Another Constructor**
  - The keyword `this` refers to the implicit parter of a method. However if the first statement of a constructor has the form

```
this(…);
```

  then the constructor calls another constructor of the same class that matches the parameter signature.

  - Using the `this` keyword in this manner is useful—you only need to write common construction code once which you factor out into a base constructor.

- **Initialisation Blocks**
  - So the two ways of initialising a data field is by setting the value in a constructor or by assigning a value in the declaration.
  - The third mechanism in Java to initialise a data field is through an **Initialisation Block**. Class definitions can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed. *The initialisation block runs before the body of the constructor while constructing an object*.
  - This mechanism is however not very common and its usually more straightforward to put the initialisation code inside a constructor.
  - In Summary, when a constructor is called:
    - All the data fields are initialised to their default values (`0, false or null`)

  - All fields initialisers and initialisation blocks are executed, in the order in which they occur in the class declaration.
  - If the first line of constructor calls a second constructor, then the body of the second constructor is executed.
  - Then the body of the constructor is executed.

- Naturally, it is always a good idea to organise your initialisation code so that another programmer could easily understand it without having to be a language lawyer. Avoid Initialisation blocks.
- To initialise a static field, either supply an initial value or use a static initialisation block. If a static field of your class requires complex initialisation code, use a static initialisation block. That is place the code inside the initialisation block and tag with the keyword `static`. E.g

```
//static initialisation block
static {
      //Constructs a new random number generator
      Random generator = new Random();

      //returns a random integer between 0 and arg-1
      nextId = generator.nextInt(10000);
}
```

- Static initialisation occurs when the class is first loaded. Like instance fields static fields are `0, false` or `null` unless you explicitly set them to another value.
- All static field initialisers and static initialisation blocks are executed in the order in which they occur in the class definition.

- **Object Destruction and `finalize` method**
  - The most common activity in a destructor is claiming memory set aside for objects. Since Java does automatic garbage collection, manual memory reclamation is not need and so Java does not support destructors.
  - But some objects utilise a resource other than the memory, such as a file or a handle to another object that uses system resources. In this case, it is important that the resource be reclaimed and recycled when it is no longer needed.
  - You can add a `finalize` method to any class. The `finalize` method will be called before the garbage collector sweeps away the object. In practice, do not rely on the `finalize` method for recycling any resources that are in short supply—you simply cannot know when this method will be called. Avoid it.

  - If a resource needs to be closed as soon as you have finished using it, you need to manage it manually. Supply a close method that does the necessary cleanup, and call it when you are done with the object.

- **Packages**
  - Java allows you to group classes in a collection called a package. Packages are convenient for organising your work and for separating your work from code libraries provided by others.
  - Just as you have nested subdirectories on you disk, you can organise packages by using levels of nesting. All standard Java packages are inside the java and naval package hierarchies.
  - The main reason for using packages is to guarantee the uniqueness of class names. To absolutely guarantee a unique package name, use an internet domain name written in reverse. You then use sub packages for different projects.
  - From the compiler point of view there is absolutely no relationship between nested packages. Each is its own independent collection of classes.

- **Class Importation**
  - A class can use all the classes from its own package and all public classes from other packages. You can access the public classes in another package in two ways. First, simply add the full package name in front of every class. Second, use the import statement to get a shorthand to refer to the classes. Once you import you no longer have to give the full name
  - You can import a specific class or the whole package. However if you import classes explicitly, the reader of your code knows exactly which classes you use.
  - You can use the * notation to import all the contents of a single package.
  - Most of the time you just import the packages that you need without worrying too much about them. The only time you need to pay attention to packages is when you have a name conflict. In such a case you get a compile time error.
  - You can solve this problem by adding a specific `import` statement to specify from which of the packages offering the same class name class do you want to import that class from.
  - In case when you need multiple classes with same name from different packages, you need to use the full package name with every class name. Locating the classes in packages is an activity of the compiler. The byte codes in the class file always use full package names to refer to other classes.

- **Static Imports**

- A form of the `import` statement permits the importing of static methods and fields, not just classes. E.g. if you add the directive

  ```
  import static java.lang.System.*;
  ```

  to the top of your source file, then you can use the static methods and fields of the System class without the class name prefix. i.e. `out.println` instead of `System.out.println`.

- You can also import a specific static method or static field using import static

- **Addition of a Class into a Package**
  - To place classes inside a package, you must put the name of the package at the top of your source file, before the code that defines classes in the package.
  - If you don't put a `package` statement in the source file, then the classes in that source file belong to the default package (which has no name) E.g put this as the first line in a source file

    ```
    package com.facebook.ankitkmr
    ```

  - Place source files into subdirectory that matches the full package name. The compiler places the class files into the same directory structure.
  - ***To compile and run classes, you must switch to and run commands from the base directory - that is the directory containing*** `com` ***directory.***

    ```
    javac com/facebook/PayrollApp.java

    java com.facebook.PayrollApp
    ```

  - The compiler operates on files(with files separators and an extension .java), whereas the Java interpreter loads a class (with dot separators).
  - You can compile the file even if it is not contained in a subdirectory `com/facebook`. The source file will compile without errors *if it doesn't depend on other packages*. However, the resulting program will not run unless you first move all class files to the right place. The *virtual machine* won't find the classes if the packages don't match the directories.

- **Package Scope**
  - Features tagged as public can be used by any class. Private features can be used only by the class that defines them. If you don't specify either public or private, the feature (that is the class, method or field) can be accessed by all the methods in the same package.
  - Good code design suggests that Variables must explicitly be marked private or they will default to being package visible. This breaks encapsulation.

Tuesday, 7 March 2017

- **The Class Path**
  - Classes are stored in the subdirectories of the file system. The path to the class must match the package name.
  - Class feels can also be stored in a JAR file. A JAR file contains multiple class files and sub directories in a compressed format, saving space, and improving performance. When you use a third party library in your program, you will usually be given one or more JAR files to include.
  - To share classes among programs, you need to do the following:
    - Place your class files inside a directory, example: `/home/user/classdir`, then this is the base directory for the package tree
    - Place any JAR files inside a directory, example `/home/user/archives`
    - Set the class path. The class path is the collection of all locations that can contain class files. In UNIX, the elements on the class path are separated by colons.
    - Example: `/home/user/classdir:.:/home/user/archives/archive.jar`, the period denotes the current directory.
    - Starting with Java SE6 you can specify a wildcard for a JAR file directory. In UNIX, the `*` must be escaped to prevent shell expansion. example: `/home/user/archives/\*`
    - The runtime library files are always searched for classes so don't include them in the class path.
  - The Class Path lists all the directories and archive files that are starting points for locating classes.
  - While searching for the classes in the runtime libraries and the locations listed in the class path, the compiler goes one step further. It looks at the source field to see if the source is newer than the class file. If so, the source file is recompiled automatically.
  - You can only import public classes from other packages. A source file can contain only one public class and the name of the file and the public class must match. Therefore the complier is easily able to locate source files for public classes. However you can import non public classes from the current package. These classes may be defined in source files with different names. If you import ta class from the current package, the compiler searched all the source files of the current package to see which one defines the class.
- **Setting the Class Path**
  - It is best to specify the class path with the `-classpath` (or `-cp`) option. The entire command must be typed into a single line. It is a good idea to place such a ling command line into a shell script or batch file.

```
java -classpath /home/user/classdir:.:/home/user/archives/
archive.jar MyProg
```

- Using the `-classpath` option to set the class path is the preferred approach for setting the class path.

- **Documentation Comments**
  - The JDK contains a very useful tool, called javadoc, that generates HTML documentation from your source files. If you add comments that start with the special delimiter /** to your source code, you too can easily produce professional-looking documentation. This is a very nice approach because it lets you keep your code and documentation in one place. If you put your documentation into a separate file, then, as you probably know, the code and comments tend to diverge over time. When documentation comments are in the same file as the source code, it is an easy matter to update both and run javadoc again.
  - The class comment must be placed after any import statements, directly before the class definition.
  - Each method comment must immediately precede the method that it describes.
  - You only need to document public fields—generally that means static constants
  - For more on documentation check out the Core Java Vol1 book, Pg, 194 onwards.

- **Class Design Hints**
  - **Always keep fields private**, doing anything else violates encapsulation. When data are kept private changes to their representation will not affect the user of the class and bugs are easier to detect.
  - **Always initialise fields**, Java won't initialise local variables for you. It will only initialise the instance fields but still don't rely on the defaults and initialise the fields explicitly.
  - **Don't use too many basic types in a class**. The idea is to replace multiple related uses of basic types with other classes. This keeps our classes easier to understand and to change. Like putting all the address fields in an `Address` class and including that in `UserProfile` class.
  - **Not all fields need individual field accessor and mutator**. Example: You won't need a setter to change the hiring date once the object is constructed.
  - **Break up classes that have too many responsibilities**. If there is an obvious way to break one complicated class into two classes that are conceptually simpler, do it.
  - **Make the names of your classes and methods and fields reflect their responsibilities.** Variables should have meaningful names that reflect what they represent. Class can be a noun.
  - **Prefer immutable classes**. The problem with mutation is that if it happens concurrently then we can have race conditions. When classes are immutable its safe to share them among threads. Computations on objects of these classes can yields new objects instead of updating the current one.

Tuesday, 7 March 2017

- **Google Java Style Guide :** **https://google.github.io/styleguide/javaguide.html**

==
==

- Inheritance

- When you inherit from an existing class, you inherit its method, and you can add new methods and fields to adapt your new class to new situations.
- **Reflection** is the ability to find out more about classes and their properties in a running program.

- Classes, Superclasses and Subclasses

- The "is-a" relationship is the hallmark of inheritance. Use the java keyword extends to denote inheritance. Like

```
public class Manager extends Employee{

        //added methods and fields

}
```

- **Defining Subclasses**
  - The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the superclass, base class or parent class. The new class is called the subclass, derived class, or child class.
  - Subclasses have more functionality than their superclasses. Even though the methods of the parent class are not explicitly defined in the subclasses, the public methods of the superclasses are automatically inherited by the subclasses.
  - Similarly all the fields public or private are also inherited (although the subclass can only access private fields through inherited methods and not directly).
  - When defining a subclass by extending its superclass, you only need to indicate the differences between the subclass and the superclass. When designing classes, you place the most general methods is the superclass and more specialised methods in its subclasses. Factoring out common functionality by moving it to a superclass is common in object oriented programming.
- **Overriding Methods**
  - Some of the superclass methods may not be appropriate for the subclasses. In such a case you need to supply a new method to override the superclass method.

- When supplying method definition for an overriding method we need to access a superclass private field. But since only the inherited methods can access the inherited fields, call the `super.getFieldName()` method to get a fields value or to set the fields value call the `super.setFieldName(…)`. Here `super` is required only if the name of the method being supplied is the same as the name of the inherited method being called in the new method definition. Otherwise we can omit `super` keyword.
- One thing to remember is that a subclass can add fields and it can add methods. It can even override existing methods of the superclass. However, inheritance can never take away any fields or methods.

• **Subclass Constructors**

- The keyword `super` has a different meaning when used as the first statement in a subclass constructor definition. The instruction:

```
super(…args…);
```

  means call the superclass constructor with the implied signature and pass it the given arguments.
- Since the subclass cant access the superclass private fields it must initialise them during subclass object construction by making a call to the superclass constructor and passing it the values that the subclass object wants for the superclass fields. The superclass constructor in this case is invoked with the `super` keyword in the subclass constructor definition and must be the first statement of all subclass constructors.
- If the subclass constructor doesn't call the superclass constructor explicitly (by means of a call to super as the first statement), the no-argument constructor of the superclass is invoked automatically. If the superclass does not have a no-argument constructor and the subclass constructor does not call another superclass constructor explicitly, the java compiler reports an error.
- `this` keyword is used to denote a reference to the implicit parameter of a method and to call another constructor of the same class.
- Likewise the `super` keyword is used to invoke a superclass method and to invoke a superclass constructor.
- The constructor call can only occur as the first statement in another constructor. The constructor arguments in such a case are either passed onto another constructor of the same class(when invoked with `this`) or to a constructor of a superclass(when invoked with `super`)
- The virtual machine knows about the actual type of object to which an object reference variable refers to and hence it can invoke the correct overridden method when it is invoked on the object.
- The fact that an object variable can refer to multiple actual types is called polymorphism. Automatically selecting the appropriate method at runtime is called dynamic binding.

- Substituting a subclass object where a superclass object is expected is possible because of polymorphism.

- **Inheritance Hierarchies**
  - Inheritance need not stop at deriving one layer of classes. The collection of all classes extending a common superclass is called an inheritance hierarchy. The path from a particular class to its ancestors in the inheritance hierarchy is its inheritance chain.

- **Polymorphism**
  - The “is-a” rule states that every object of the subclass is an object of the superclass. Hence, you can use a subclass object whenever the program expects a superclass object.
  - In Java, object variables are polymorphic. A variable of type X can refer to an object of X or to an object of any subclass of the X.
  - However when a subclass object is assigned to a superclass object variable, the only methods that can be invoked on it are the ones that can be invoked on an object of type the variable is of. So in the following:

```
Employee[] staff = new Employee[3]();

staff[0] = new Manager(); //extends Employee

staff[0].setBonus(2000); // would be compile time error

//only the methods that can be invoked on Employee objects
can be invoked on subclass objects of Employee referred to
by an Employee reference variable.
```

  - Also, the reverse is not possible, you cannot assign a superclass object to a subclass reference variable.
  - In Java, arrays of subclass references can be converted to arrays of superclass references without a cast. But in order to prevent a program from converting a subclass reference to superclass reference and then storing a superclass object in it (as that would mean a reference variable of actual type that is a subclass referring to a superclass object), all arrays rememberer the element type with which they were created, and they monitor that only compatible references are stored into them. Attempting to store a superclass object to a subclass array after converting to superclass array throws a `ArrayStoreException`.

- **Understanding Method Calls**
  - This is how a method call is applied to an object:
    - The compiler looks at the declared type of the object and the method name. The compiler enumerates all the methods with the same name in the Class of which the Variable has been declared to be reference of and all accessible (public) methods with

same name in the superclasses of the Class the variable is declared reference of. Now the compiler knows all the possible candidates for the method to be called.

- Next, the compiler determines the types of the arguments passed in the method call. If among all the methods enumerated above there is a unique method whose parameter types(signature) are a best match for the supplied arguments, that method is chosen to be called. This process is called overloading resolution. The compiler now knows the name and parameter types of the method to be called.
- Note: If you define a method in the subclass that has the same method signature as a method in the superclass, you override the superclass method. Method signature includes the method name and the parameter order and types. Return type is not a part of the method signature, however when you overrides a method, you need to keep the return type covariant to the original return type. That is, a subclass method overriding a superclass method can change the return type to a subtype of the original type. The two methods in this case are said to have **covariant return types**.
- If the method is private, static, final or a constructor, then the compiler knows exactly which method to call. This is called static binding. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtime. In such a case the compiler would issue an instruction to invoke the method in the method call on object with dynamic binding during runtime.
- When the program runs and sues dynamic binding to call a method, the virtual machine must call the version of the method that is appropriate for the actual type of the object on which the method is being invoked. The method is searched for in a bottom-up approach from the actual class to its superclass and so on.
- It would actually be very time consuming to carry out this search every time a method is called. Therefore the virtual machine precomputes for each class a method table that lists all method signatures and the actual methods to be called. When a method is actually called, the virtual machine simply makes a table lookup. If the call is `super.methodName(…args…)`, then the virtual machine consults the method table of the superclass of the implicit parameter.

- At runtime the call to a method gets resolved as follows:
  - First the virtual machine fetches the method table for the actual type of the object. So the method table of a class is a list of all the methods that can be invoked on the object of that class and their corresponding method.
  - Then, the virtual machine looks up the defining class for the method with appropriatee method signature. Now it know which method to call.
  - Finally the virtual machine calls the method.
- Dynamic binding makes programs extensible without the need for modifying existing code.

- When you override a method, the subclass method must be atleast as visible as the superclass method.

- **Preventing Inheritance: Final Classes and Methods**
    - Occasionally you may want to prevent someone from forming a subclass from one of your classes. Classes that cannot be extended are called **final classes,** and you use the `final` modifier in the definition of the class to indicate this.
    - You can also make a specific method in a class `final`. If you do this, then no subclass can override that method. (All methods in a final class are automatically `final`)
    - There is only one good reason to make a method or a class final: to make sure its semantics cannot be changed in a subclass.
    - Some programmers believe that you should declare all methods as final unless you have a good reason to want polymorphism. If a method is not overridden, and its short, then the compiler can optimise the method call away—a process called inlining.
    - Inlining is good because CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one.
    - But if a method can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the override code might do.
    - Fortunately, the just-in0time compiler in the virtual machine can do a better job than traditional compilers. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called and not actually overrides, the just-in-time compiler can inline the method.
    - If the virtual machine, then loads another subclass that overrides an inlined method, then the optimiser must undo the inlining. This takes time but it happens rarely.

- **Casting**
    - The process of forcing a conversion form one type to another is called casting. You may need to convert an object reference from one class to another.
    - To cast a reference variable of one class to another, surround the target class to which you want to cast the reference to and place it in front of the object reference to be casted.
    - There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten.
    - In Java, every variable has a type. The type of an object reference variable describes the kind of object the variable refers to and what it can do.
    - The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass object reference to a superclass variable, you are

promising less, and the compiler will simply let you do it. If you assign a superclass reference to a subclass variable, then you'd be promising more. In such a case you must use a cast so that your promise can be checked at runtime.

```
//cast to another class reference
Manager boss = (Manager) staff[0];
```

- When the program runs, the Java runtime system checks if the cast is valid and in case of an incorrect cast it generates a `ClassCastException`. If you do not catch that exception, your program terminates. Thus it is a good programming practice to find out if a cast will succeed before attempting it. Simply use the `instanceof` operator. Example:

```
if ( staff[1] instanceof Manager){

     boss = (Manager) staff[1];

}
```

- Finally the compiler will not let you make a cast if there is no chance for the cast to succeed. Example

```
String c = (String) staff[1];
```

is compile-time error because `String` is not a subclass of `Employee` class.

- To sum up:
  - You can cast only within an inheritance hierarchy
  - Use `instanceof` operator to check before casting form a superclass to a subclass.
- The test `x  instanceof  C;` does not generate an exception if `x` is `null`. It simply returns `false`.
- Usually there is no need to make a cast as dynamic binding that makes polymorphism work locates the correct method automatically. The only reason to make a cast is to use a method that is unique to the subclass.
- Remember it takes only one uncaught ClassCastException to terminate your program. In general is best to minimise the use of cast and `instanceof` operator.

- **Abstract Classes**
  - As you move up the inheritance hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes so general that you think of it more as a basis for other classes than as a class with specific instances you want to use.
  - We can use this class to specify common fields and declare common methods. Sometimes while the method might be common to subclasses, its implementation can be subclass-dependent. In such cases, if you use the abstract keyword, then you do not need to implement that method. Example:

- A class with one or more abstract methods must itself be declared to be abstract. Such a class is then called an abstract class. Methods declared in an abstract class do not need to be implemented and that class can't be instantiated i.e. we cant have objects of that class. In addition to abstract methods, abstract classes can have fields and concrete methods.

```
public abstract class Person{
     private String name;

     // abstract classes can have constructors
     public Person(String name){
          this.name = name;
     }
     .. .. .. ..
     public abstract String getDescription();
     // no implementation required as declared
     // to be abstract
}
```

- The abstract methods act as placeholders for methods that are implemented in the subclasses. The subclasses inherit the fields and the methods declared and if they are concrete classes then they need to provide implementation for the methods as it suits them.
- So when you extend an abstract class, you have two choice. You can leave some or all of the abstract methods undefined; then you must tag the subclass as abstract as well. Or you can define all methods, and the subclass is no longer abstract.
- A class can even be declared as abstract even though it has no abstract methods. Like I said, abstract classes cannot be instantiated. So if you don't want anyone to create an object of your class, you declare it as abstract. ***Still, abstract classes can have constructors that can be called with*** `super` ***in constructors of the subclasses.***
- However while you cant have objects of abstract classes, you can still create object reference variables of an abstract class, but such a variable must refer to an object of a concrete subclass.
- The reason for declaring an abstract method in an abstract class is that it can act as a placeholder. As the compiler ensures that you invoke only methods that are declared in the class. This helps to ensure that when applying polymorphism by assigning subclass references to abstract class variables, the common methods can still be called.

• **Protected Access**

- Fields in a class are best tagged as `private`, and methods usually tagged as `public`. Any features tagged `private` won't be visible to other classes including the subclasses.
- However, when you want to restrict access to a method to the subclasses only or less commonly to allow subclass methods to access a superclass fields, you can declare the class feature as protected.

- Protected features are then accessible to subclasses. However, unlike class based privilege access, a subclass method can only access the superclass field of another object of the same subclass. That is if `hireDay` field in `Employee` class is protected, then `Manager` (extends Employee) methods can access it. But also those methods can only access the same field of other objects that belong to the same subclass `Manager` and not of any object of `Employee` class.
- **This restriction is made so that you cant abuse the protected mechanism by forming subclasses just to gain access to protected fields.**
- In practice, use `protected` fields with caution. Suppose your class is used by other programmers and you designed it with `protected` fields. Unknown to you, other programmers may inherit classes from your class and start accessing your protected fields. In this case you can no longer change the implementation of your class.
- `Protected` methods make more sense. a class may declare a method as protected is it is tricky to use. this indicates that the subclass can be trusted to use the method correctly, but other classes cannot.
- Summary of Access Modifiers in Java that control visibility.
  - Visible to the class only - `private`
  - visible to the whole world - `public`
  - Visible to the package and all subclasses - `protected`
  - Visible to the package. No modifiers needed. Default.

- **Object: The Cosmic Superclass**
  - The `Object` class is the ultimate ancestor—every class in Java extends `Object`. If no superclass is explicitly mentioned in a class definition, it is taken for granted that it extends `Object` class.
  - Since every class extends `Object` class and polymorphism states that an object of subclass can be substituted where an object of superclass is expected, **you can use a variable of type `Object` to refer to objects of any type.**
  - But a variable of type Object is only useful as a generic holder for arbitrary values. To do anything specific with the value, you need to have some knowledge about the original type and apply a cast. Like

```
Employee e = (Employee) obj;
```

  - In Java, only the primitive values (numbers, characters and boolean values) are not objects. All array types no matter whether they are arrays of objects or of primitive types, are class types that extend `Object` class.
  - **The `equals` Method**

- The `equals` method in the `Object` class tests whether one object is considered equal to another. The equals method, as implemented in the `Object` class, determines whether two object references are identical — if two objects are identical, they should be equal.
- Remember the `equals` method signature defines an `Object` type as the explicit parameter. So when defining a new method to override the `equals` method define the explicit parameter to be of type `Object`.
- However you will often want to implement **state-based equality testing**, in which two objects are considered equal when they have the same state.
- the `getClass()` method returns the class of an object. (I*n our tests, two objects can be equal only when they belong to the same class*. *You may decide not to let this be a constraint*)
- When you define the `equals` method for a subclass to implement state-based equality, first call `equals` on the superclass. If that test doesn't pass, then the objects can't be equal. If the superclass fields are equal, you are ready to compare the instance fields of the subclass.

• **Equality Testing and Inheritance**

- Although instead of checking for class equality (through `getClass()`), some programmers use `instanceof` test. This leaves open the possibility that other object can belong to a subclass
- The Java Language Specification requires that the `equals` method has the following properties:
  - It is *reflexive*: For any non-null reference x, x.equals(x) should return true.
  - It is *symmetric*: For any references x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
  - It is *transitive*: For any references x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
  - It is *consistent*: If the objects to which x and y refer haven't changed, then repeated calls to x.equals(y) return the same value.
  - For any non-null reference x, x.equals(null) should return false.

- There are two distinct scenarios:
  - If subclasses can have their own notions of equality, then the symmetry requirement forces you to use getClass test.
  - If the notion of equality testing is fixed in the superclass, then you can use the instanceof test and allow objects of different subclasses to be equal to one another if they have the same values for the fields of the common superclass in which the notion

of equality is fixed. But in this case you also should then **declare the `equals` method in the superclass where the equality notion is fixed as `final`.**

- **Here's the recipe for writing a perfect `equals` method:**

  1. Name the explicit parameter `otherObject`—later, you will need to cast it to another variable that you should call `other`.

  2. Test whether `this` happens to be identical to `otherObject`. This statement is just an optimisation. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields:

     ```
     if (this == otherObject) return true;
     ```

  3. Test whether `otherObject` is null and return `false` if it is. This test is required.

     ```
     if (otherObject == null) return false;
     ```

  4. Compare the classes of `this` and `otherObject`. If the semantics of equals can change in subclasses, use the `getClass` test:

     ```
     if (getClass() != otherObject.getClass())
          return false;
     ```

     If the same semantics holds for *all* subclasses, you can use an `instanceof` test:

     ```
     if (!(otherObject instanceof ClassName)) return false;
     ```

  5. Cast `otherObject` to a variable of your class type:

     ```
     ClassName other = (ClassName) otherObject
     ```

  6. Now compare the fields, as required by your notion of equality. Use == for primitive type fields, `Objects.equals` for object fields. If you have fields of array type, you can use the static Arrays.equals method to check that the corresponding array elements are equal. Return `true` if all fields match, `false` otherwise.

     ```
     return field1 == other.field1 &&
     Objects.equals(field2, other.field2) &&...;
     ```

     If you redefine equals in a subclass, include a call to super.equals(other).

- Make sure to tag the methods that you are defining with the intention of overriding a superclass method with `@Override` annotation. This way if you make a mistake in the method signature and hence end up defining a new method instead of overriding an existing one, the compiler reports an error.

• **The `hashCode` Method**

- A hash code is an integer that is derived from an object. Hash codes should be scrambled—if x and y are two distinct objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different.

- The String class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
   hash = 31 * hash + charAt(i);
```

- The hashCode method is defined in the Object class. Therefore every object has a default hashCode. That hashCode is derived from the objects memory location.
- As shown above, the hashCode for strings are derived from their contents, whereas if you take the StringBuilder class objects their hash codes is derived from their memory location, hence two string objects with same content will have the same hashCode whereas two StringBuilder objects with the same contents will have different hashCode since their memory location will be different.
- **If you redefine the `equals` method, you will also need to redefine the `hashCode` method for objects that the users might insert into a hash table**. As a rule of thumb whenever you redefine `equals`, redefine `hashCode`. Moreover your definition for `hashCode` has to be compatible with your `equals` definition. That is **if the `equals` method returns two objects to be equal then there `hashCode` should also be equal.** *This means, while supplying definition for computing the object's hashCode, only include the hashCodes of the instance fields that are considered while deciding equality of two objects of a particular class.*
- **The `hashCode` method should return an integer (which can be negative). For computing the hashCode, combine the `hashCodes` of the instance fields so that the `hashCode` for different objects are likely widely scattered.**
- Use the null-safe method `Objects.hashCode` to get hashCodes of object instance fields. It returns `0` if the argument is `null` and the result of calling `hashCode` on the argument otherwise. Also use the static `Double.hashCode` method to avoid creating a `Double` object.
- **Even better, when you need to combine multiple hash values, `Objects.hash` with all of them. It will call `Objects.hashCode` for each argument and combine the values**. **pay attention the class used here is `Objects` and not `Object`.

```
@Override
public int hashCode() {
   return Objects.hash(name, salary, hireDay);
}
```

- If you have fields of an array type, you can use the static `Arrays.hashCode` method to compute a hash code composed of the hash codes of the array elements.

• **The `toString` Method**

- The `toString` method of the `Object` class returns a string representing the value of the object on which it is invoked.

- Most toString methods follow the patter of returning: the ClassName, then the field values enclosed in square brackets. (Use `getClass().getName()` to get the name of the class instead of hardwiring it in the string as in case of getClass, the subclasses can call the `super.toString()` and concatenate the returned string with the value of subclass instance fields.

```
public class Manager extends Employee {
   public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
   }
}
```

- Whenever an object is concatenated with a string by the “+” operator, the Java compiler automatically invokes the toString method to obtain string representation of the object. If you pass an object to `System.out.println` method call, then the `println` method simply calls the `toString` method on the object.
- The `Object` class defines the `toString` method to return the name of the class and the hashCode of the object. Annoyingly, array object inherit this method from `Object` class and so In order to print arrays meaningfully, use `Arrays.toString` method instead.
- We strongly recommend that you add a toString method to each class that you write.
- Just like `getClass()` you have `getSuperClass()` method to get the Class object of the superclass of the current class.

- **Generic Array Lists**
    - In Java, you can set the size of an array at runtime which is better than having to fix them at compile time as happens in C++. It still does not solve the problem of dynamically modifying arrays at runtime. Once you set the array size, you can not change it.
    - In order to be able to dynamically modify the size during runtime, use `ArrayList`. `ArrayList` is a data structure that automatically adjusts its capacity as you add and remove elements, without any additional code.
    - `ArrayList` is a generic class with a type parameter. To specify the type of the elements of the objects that the array list holds, yo upped the class name enclosed in angle brackets such as:

```
ArrayList<Employee> staff =  new ArrayList<>();
```

    - As you see above, we can omit the type in the angle brackets on the right side. The compiler checks what happens to the new value. If it is assigned to a variable, passed into a method or returned from a method, then the compiler checks the generic type of the variable, parameter or method. It then places that type into the <>
    - Use the add method to add new elements to an ArrayList. The array list manages an internal array of object references. Eventually, that array will run out of space. This is

where array lists work their magic: If you call add and the internal array is full, the array list automatically creates a bigger array and copies all the objects from the smaller to the bigger array.

- If you already know, or have a good guess, how many elements you want to store, call the ensureCapacity method before filling the array list:

  ```
  staff.ensureCapacity(100);
  ```

  That call allocates an internal array of 100 objects. Then, the first 100 calls to add will not involve any costly reallocation. You can also pass an initial capacity to the ArrayList constructor:

  ```
  ArrayList<Employee> staff = new ArrayList<>(100);
  ```

- The `size()` method returns the actual number of elements in the array list.
- Once you are reasonably sure that the array list is at its permanent size, you can call the `trimToSize` method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory. Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use `trimToSize` when you are sure you won't add any more elements to the array list.

• **Accessing ArrayList Elements**

- In order to access or set the element of an array, you use the `get` and `set` methods. (Index values are zero based). Use the `add` method instead of `set` to fill up an array, and use `set` only to replace a previously added element.
- In order to get flexible growth and convenient element access (via indexing and not methods) first make an array list and add all the elements. When you are done, use the toArray method to copy the elements into an array. Example:

  ```
  //ArrayList values copied to the array passed
  staff.toArray(sampleArray);
  staff.add(i, objectToInsertInMiddleAtIndex);
  staff.remove(i);
  ```

- Inserting and removing elements is not terribly efficient. if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead.
- You can use for each loop to traverse the contents of the ArrayList:

  ```
  for (Employee e: staff){
        //do something with e
  }
  ```

• **Compatibility between Typed and raw ArrayLists**

- **Always use type parameters with generic classes for added safety.**

- One thing to remember is that the type parameter in Generic classes is a new addition to the Java language so you'll often find legacy code that does not use type parameters.
- You can pass a typed array list where raw array list is expected. Conversely, you can pass raw array list to a typed one but this time you get a warning and using a cast doesn't make the warning go away.
- For compatibility, the compiler translates all typed array lists into raw ArrayList objects after checking that the type rules were not violated. In a running program all array lists are same—there are no type parameters in the virtual machine. Thus the casts `(ArrayList)` and `(ArrayList<Employee>)` carry out identical runtime checks.
- When you interact with legacy code, study the compiler warnings and satisfy yourself that the warnings are not serious.

- **Object Wrappers and Autoboxing**
  - Occasionally you may need to convert a primitive type into an object. In Java, All primitive types have class counterparts. These classes are called **wrappers**. The wrapper classes are: `Integer`, `Double`, `Long`, `Float`, `Short`, `Byte`, `Character` and `Boolean`.
  - The wrapper classes are immutable—you cannot change the value after the wrapper has been constructed. They are also final so you cannot subclass them.
  - A sample use of wrapper classes is when we need to declare array lists. Since array lists can only store objects, we wrap the primitive values and store them in the array list.

```
ArrayList<Integer> list = new ArrayList<>();
```

  - When you pass an `int` value where `Integer` type is expected the `int` value is automatically boxed or autoboxed into an `Integer` object by the compiler and then passed. Conversely, if you pass an `Integer` object where an `int` is expected, the `Integer` object is automatically unboxed and it's wrapped `int` value passed by the compiler. Automatic boxing and unboxing even works with arithmetic expressions.

```
list.add(3);    //becomes
list.add(Integer.valueOf(3));

int n = list.get(i);       //becomes
int n = list.get(i).intValue();

Integer n = 3;
n++; //becomes
n = Integer.valueOf(n.intValue()+1);
```

The compiler automatically inserts instructions to unbox the object, increment the resulting value, and box it back.

- In most cases, it appears that the primitive types and their wrappers are one and same. But they differ when tested for equality using == operator as in case of wrappers this operator checks for their identical memory locations and not the state based test. The remedy is to call equals method when comparing wrapper objects.
- Since wrapper class references can be null, it is possible for auto-unboxing to throw a NullPointerException.

```
Integer n = null;
System.out.println(2 * n); // Throws NullPointerException
```

- Also, if you mix Integer and Double types in a conditional expression, then the Integer value is unboxed, promoted to double, and boxed into a Double.

```
Integer n = 1;
Double x = 2.0;
System.out.println(true ? n : x); // Prints 1.0
```

- Boxing and unboxing is a courtesy of the *compiler*, not the virtual machine. The compiler inserts the necessary calls when it generates the bytecodes of a class. The virtual machine simply executes those bytecodes.
- The designers of Java found the wrappers a convenient place to put certain basic methods, such as those for converting strings of digits to numbers.

```
Integer.parseInt(S); //converts digit strings to int
```

- **Methods with Variable Number of Parameters**
  - It is possible to provide methods that can be called with a variable number of parameters. For example, `System.out.printf` method takes in variable number of arguments. You can define your own methods with variable parameters, and you can specify any type for the parameters, even a primitive type
  - **To denote that a method takes variable parameters use ellipsis (three consecutive dots) or the** … after the type of the parameters and before the name of the array they'll be put into by the compiler.
  - If the method can take variable parameters of different object types, use

```
//notice three dots after parameter type
(Object… args)
//else for same types use
(Employee… staff) or (double… values).

//Like in the example below

public static double max(double... values) {
     double largest = Double.NEGATIVE_INFINITY;
     for (double v : values){
          if (v > largest){
```

```
                    largest = v;
                }
            }
            return largest;
        }
```

- **The variable parameters are put in an array of the defined element type and passed to the method**. So it is also legal to pass an array of the declared element type as the last parameter of a method with variable parameters. Therefore, you can redefine an existing method whose last parameter is an array to a method with variable parameters, without breaking any existing code.

- **Enumeration Classes**

```
        public enum Size {
            SMALL("S"),
            MEDIUM("M"),
            LARGE("L"),
            EXTRA_LARGE("XL");

            private String abbreviation;

            private Size(String abbreviation){
                this.abbreviation = abbreviation;
            }

            public String getAbbreviation(){
                return abbreviation;
            }
        }

        //or you can also simply define it as below

        public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

- This is how you define an enumerated type. The type defined by this declaration is actually a class. The class has exactly four instances—it is not possible to construct any new objects. Therefore you can simply check for equality of enumerated type using == operator.
- You can add constructors, methods and fields to an enumerated type as shown above. The constructors are only invoked when the enumerated constants are constructed.
- All enumerated types are subclasses of the class `Enum`. They inherit a number of methods from that class. The most important one is toString which returns the name of the enumerated constant. For example, `Size.SMALL.toString()` returns the string `“SMALL”`.
- The converse of toString method is the valueOf method that returns the enumerated type instance with the corresponding name.

```
    //sets s to Size.SMALL
    Size s = Enum.valueOf(Size.class, "SMALL");
```

- Each enumerated type has a static values method that returns the array of all values of the enumeration. For example:

```
    Size[] values = Size.values();
```

  returns the array with elements`[Size.SMALL, Size.MEDIUM, Size.LARGE, and Size.EXTRA_LARGE].`

- The ordinal method yields the position of an enumerated constant in the enum declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns 1.

- The example below demonstrates how to work with enumerated constants.

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

          Scanner in = new Scanner(System.in);
          System.out.print("Enter a size: (SMALL,
                              MEDIUM, LARGE, EXTRA_LARGE) ");

          String input = in.next().toUpperCase();
          Size size = Enum.valueOf(Size.class, input);
          //toString() called automatically
          System.out.println("size=" + size);
          System.out.println("abbreviation=" +
                              size.getAbbreviation());
          if (size == Size.EXTRA_LARGE){
               System.out.println("Good job--you paid
                                   attention to the _.");
          }
     }
}
```

- **Reflection**

  - The reflection library gives a very rich and elaborate toolset to write programs that manipulate Java code dynamically. In particular, when new classes are added at design time or runtime, rapid application development tools can dynamically. inquire about the capabilities of these classes. For example like while using Spring Framework

  - A program that can analyse the capabilities of classes is called reflective. You can use reflection to:

    • Analyse the capabilities of classes at runtime;

- Inspect objects at runtime—for example, to write a single toString method that works for *all* classes;
- Implement generic array manipulation code; and
- Take advantage of Method objects that work just like function pointers in languages such as C++.

- **The `Class` Class**
  - While your program is running, the Java runtime system always maintains what is called **runtime type identification** on all objects. This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute.
  - You can also access this information by working with special Java class called `Class`. The `getClass()` method in the Object class returns an instance of this `Class` type.

```
//Assume values assigned to argument variables
Employee e = new Employee(name, salary, hireDay);
Class myClass = e.getClass();
```

  - Just like an employee object described the state of a particular employee, a `Class` object describes the properties of a particular class. The most commonly used method of `Class` is `getName`. It returns the name of the class. If the class is in a package, then the package name is a part of the class name.
  - You can even obtain a `Class` object corresponding to a class name by using the static `forName` method.

```
String className = "java.util.Random";
Class cl = Class.forName(className);
```

  - Use this method if the class name is stored in a string that varies at runtime. This works if the `className` variable above is the name of a class or an interface. Otherwise this method throws a checked exception.
  - Another shorthand for obtaining a `Class` object is : If `T` is an Java type, then `T.class` is the corresponding class object.
  - A `Class` object really describes a type, which may or may not be a class. For example, `int` is not a class, but `int.class` is an object of type `Class`. The `Class` class is actually a generic class. So `Employee.class` is of type `Class<Employee>`.
  - **Caution**: For historical reasons, the `getName` method returns somewhat strange names for array types:
  - The virtual machine manages a unique `Class` object for each type. Therefore, you can use the == operator to compare class objects.

- Another useful method is `newInstance()` which creates a new instance of the class type it is invoked on by calling that class' no-argument constructor. If that class does not have a no-argument constructor an exception is thrown. A combination of forName and newInstance lets you create an instance of a class name stored in a string.

```
Employee e = new Employee();
String s = “com.mydomain.myclasses.Employee”;
Employee e2 = e.getClass().newInstance();
Employee e3 = Class.forName(s).newInstance();
```

- If you need to provide parameters for the constructor of a class you want to create by name in this manner, you can't use the above statements. Instead, you must use the newInstance method in the Constructor class.

• **A Primer on Catching Exceptions**

- When an error occurs at runtime, a program can throw an exception. Throwing an exception is more flexible than terminating the program because you can provide a handler that catches the exception and deal with it.
- If you don't provide a handler, the program still terminates and prints the message to the console, giving the type of the exception.
- There are two kinds of exceptions: **unchecked exceptions** and **checked exception**. With checked exception, the compiler checks that you provide a handler. However many common exceptions are unchecked. The compiler does not check whether you provide a handler for these errors because ideally you should spend mental energy on avoiding these mistakes and not on writing handlers for them. But not all errors are avoidable. If an exception can occur despite best efforts, then the compiler insists that you provide a handler. Hence the case for checked exception.
- Place one or more statements that might throw checked exception inside a `try` block. Then provide the handler code in the `catch` clause.

```
try{
//statements that might throw exceptions            }catch
(Exception e){                                       /   /
handler action                                   }
```

- If an exception occurs, then the rest of the statements in the try block are skipped and the control moves to handler code. On the other hand, if none of the statements in the try code throw an exception then the code in the handler block is skipped. You only need to provide handlers for checked exception.

• **Using Reflection to Analyse the Capabilities of Classes**

- The three classes `Field, Method` and `Constructor` in the `java.lang.reflect` package describe the fields, methods and constructors of a class respectively.
- All three classes have a method called `getName` that returns the name of the item. The `Field` class has a method `getType` that returns an object of type `Class` that describes the field type. The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports and return type.
- All three of these classes also have a method called `getModifiers` that returns an integer with various bits turned on and off, that describes the modifiers used such as `public static`. You can then use the static methods in the `Modifier` class in the `java.lang.reflect` package to analyse the integer that `getModifiers` returns. Use methods like `isPublic, isPrivate` or `isFinal` in the modifier class to tell whether a method or a constructor was `public, private` or `final`.
- All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifier` returns. You can also use `Modifier.toString` method to print he modifiers.
- The `getFields`, `getMethods` and `getConstructors` methods of the `Class` class return arrays of the public fields, methods and constructors that the class supports. This includes the public members of the superclasses. The `getDeclaredFields`, `getDeclaredMethods` and `getDeclaredConstructors` methods of the `Class` class return arrays consisting of all fields, methods, and constructors that are declared in the class. This include `private, package`, and `protected` members but not members of the superclasses.
- **[IMPORTANT]** Check out **Listing 5.13 on page 267 in the Core Java Vol 1** book for the code of a program that can analyse any class. Also check out the APIs after the code.

- **Using Reflection to Analyse Objects at Runtime**
  - In the previous section we figured out how to get the names and types of fields in a class. In this section we'll look at the contents of those fields of objects at runtime.
  - The key method to achieve this is the `get` method in the `Field` class. If `f` is an object of type `Field` and `obj` is an object of the class of which `f` is a field, then `f.get(obj)` returns an object whose value is the current value of the field of `obj`.
  - You can only use `get` method to get the values of accessible fields. The security mechanism of Java lets you find out what fields an object has, but it won't let you read the values of those fields unless you have access permission.
  - In the example below, since the `name` field is a `private` field, the `get` method will throw an `IllegalAccessException`.

```
Employee harry = new Employee("Harry Hacker",
                              35000, 10, 1, 1989);
```

```
Class cl = harry.getClass();
// the class object representing Employee
Field f = cl.getDeclaredField("name");
// the name field of the Employee class
Object v = f.get(harry);
// the value of the name field of the harry object, i.e.,
// the String object "Harry Hacker"
```

- The default behaviour of the reflection mechanism is to respect Java access control. However, if a Java program is not controlled by a security manager that disallows it, you can override access control. To do this, invoke the `setAccessible` method on a `Field`, `Method`, or `Constructor` object. For example, the line of code below makes it possible to access the `private` field `name` in the above example.

```
f.setAccessible(true); //add this before get call
```

- The `setAccessible` method is a method of the `AccessibleObject` class, the common superclass of the `Field`, `Method` and `Constructor` classes. This feature is provided for debuggers, persistent storage and similar mechanisms.
- There is just one issue with the get method that it returns an object so if the field that you are trying to retrieve is a primitive then the compiler is going to autobox the value of the field in its corresponding wrapper class object and return that.
- Of course, you can also set the value of the fields that you can get. The call `f.set(obj, newValue)` sets the field represented by `f` of the object `obj` to the `newValue`.
- To write a generic `toString` method that works for any class, use `getDeclaredFields` to obtain all data fields. Then use the `setAccessible` method to make all fields accessible. For each field, obtain the name and the value. Then turn each value into a string by recursively invoking `toString`.
- The generic `toString` method needs to address a couple of complexities. Cycles of references could cause an infinite recursion. Therefore, the program needs to keep track of objects that were already visited. Also, to peek inside arrays, you need a different approach.
- [**IMPORTANT**] Checkout **Listing 5.15 on page 274 Core Java Vol 1** and check API after that code.

• **Using Reflection to Write Generic Array Code**

- The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. This is used, for example, in the implementation of the `copyOf` method in the `Arrays` class. That is why this code is so generic that it can take array of any element type and return a new array object of new length and the same element type.
- The reasons why we need reflection is that while creating the new array in the method implementation, we cannot use an `Object[]` because an array of objects cannot be cast to an array of employees. A java array remembers the type of its entries—that is, the element type used in the new expression that created it. It is legal to cast an `Employee[]` to an `Object[]` temporarily and then back to `Employee[]`, but an array that started as an `Object[]` can never be cast into an `Employee[]` array. To write this kind of generic code we need to be able to make a new array of the *same type* as the original array.
- The key to making this work is the newInstance method of the Array class that constructs a new array. We need to pass the type of the element and the desired length of the new array.
- We obtain the length by calling Array.getLength(a). The static getLength method of the Array class returns the length of the array. To get the component type of the new array:
  - Get the Class of the object of `passedArray` and confirm that it is indeed an array
  - Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.

```
public static Object goodCopyOf(Object passedArray, int newLength){
   Class cl = a.getClass();
   if (!cl.isArray()) return null;

   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   Object newArray = Array.newInstance(componentType, newLength);
   System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
   return newArray;
}
```

- Note that this `copyOf` method can be used to grow arrays of any type, not just arrays of objects. To make this possible, the parameter of `goodCopyOf` is declared to be of type `Object`, *not an array of objects* (`Object[]`). The integer array type `int[]` can be converted to an `Object`, but not to an array of objects! The calling program can then simply cast the return value of this method into the respective type of the original array that it passed.
- **Check out** `java.lang.Array` **API on page 279, Core Java Vol 1**

- **Invoking Arbitrary Methods**

- In C and C++, you can execute an arbitrary function through a function pointer. On the surface, Java does not have method pointers—that is, ways of giving the location of a method to another method, so that the second method can invoke it later. However, the reflection mechanism allows you to call arbitrary methods.
- The `Method` class has an `invoke` method that lets you call the method that is wrapped in the current `Method` object. The signature for the `invoke` method is:

```
        Object invoke(Object obj, Object... args)
```

The first parameter is the implicit parameter, and the remaining objects provide the explicit parameters. For a `static` method, the first parameter is ignored—you can set it to `null`.

- As you can see from the signature above it returns `Object` type, you need to cast the return value to the return type of the wrapped method. If the return type is a primitive type, the invoke method will return the wrapper type instead. Then you must cast the return value with the wrapper class and then use auto-unboxing to assign the return value to the original return type variable.
- To get a `Method` object, call `getDeclaredMethods` and search through the returned array of `Method` objects until you find the method you want. Or, you can call the `getMethod` method of the `Class` class. This is similar to the `getField` method that takes a string with the field name and returns a `Field` object. However, there may be several methods with the same name, so you need to be careful that you get the right one. For that reason, you must also supply the parameter types of the desired method. The signature of `getMethod` is:

```
Method getMethod(String name, Class... parameterTypes)

//example use for demonstration
Method m1 = Employee.class.getMethod(“raiseSalary”,double.class);
```

- Just as in C, this style of programming is usually quite inconvenient, and always error-prone. Also, the parameters and return values of invoke are necessarily of type Object. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code, so errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than code that simply calls methods directly.
- In particular, we echo the developers of Java and suggest not using Method objects for callback functions. Using interfaces for the callbacks leads to code that runs faster and is a lot more maintainable.

- **Design Hints for Inheritance**

- **Place common operations and fields in the superclass.**
- **Don’t use protected fields.**
  - The protected mechanism doesn’t give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses protected instance fields, thereby breaking encapsulation. And second, in the Java programming language, all classes in the same package have access to protected fields, whether or not they are subclasses.
  - `protected` *methods, on the other hand, can be useful to indicate methods that are not ready for general use and should be redefined in subclasses.*
- **Use inheritance to model the “is-a” relationship** (The subclass must be a special/more specific case of the superclass)
- **Don’t use inheritance unless all the inherited methods make sense.**
- **Don’t change the expected behaviour when you override a method.**
- **Use polymorphism, not type information**
  - Whenever you find code of the following form, think polymorphism.

    ```
    if(x is of type 1) action1(x);
    else if(x is of type 2) action2(x);
    ```

    Do *action1* and *action2* represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call `x.action();` and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.
  - Code that uses polymorphic methods or interface implementations is much easier to maintain and extend than code using multiple type tests.
- **Don’t overuse reflection**
  - The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at runtime. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications.
  - Reflection is fragile—with it, the compiler cannot help you find programming errors. Any errors are found at runtime and result in exceptions.

- Interfaces, Lambda Expressions, and Inner Classes

- Interfaces, is a way of describing what classes should do, without specifying how they should do it. A class can implement one or more interfaces. You can then use objects of one of these implementing classes whenever conformance to an interface is required.
- Lambda expressions is a way of expressing a block of code that can be executed at the later point of time.
- Inner classes are technically classes defined inside other classes, and their methods can access fields of the enclosing scope. Inner classes are useful when you design collections of cooperating classes.
- Proxies are objects that implement arbitrary interfaces. A proxy is a very specialised construct useful for building system-level tools.

- Interfaces

- **The Interface Concept**
  - An interface is not a class but a set of requirements for the classes that want to conform to the interface. Typically a supplier of some service says "If you conform to a particular interface then I'll perform the service".
  - For Example, the `sort` method of the `Arrays` class promises to sort an array of objects but under one condition that the objects belong to class that implements the `Comparable` interface. The `Comparable` interface has a single method declared:

    ```
    //for Comparable interface
    int compareTo(Object other);

    //for Comparable<Employee> interface
    int compareTo(Employee other);
    ```

    This means that any class that implements the `Comparable` interface is required to have a compareTo method, and the method must have the same signature i.e it must take, in this case, an `Object` parameter and return an integer.
  - *All methods of an interface are automatically* `public`. For that reason, it is not necessary to supply the `public` keyword when declaring a method in an interface and keeping in mind the convention to reduce redundancy, rather it is recommended to not supply the `public` keyword.
  - On the other hand, it is necessary to supply the `public` keyword when supplying the method definition in a class that implements the interface because otherwise the compiler complains that a restrictive access modifier is used.
  - Interfaces can have one or more declared methods. They can also have constants. Just like all methods in an interface are `public` by default, all fields in an interface are `public static final` by default. Some interfaces define just constants and declare no methods.

- Interfaces can never instance fields. They can have simple methods which can't access any instance fields because interfaces don't have any. And they can have static methods.
- Supplying instance fields and methods that operate on them is the job of the classes that implement the interfaces.
- To make a class implement an interface:
  - You declare that your class intends to implement the given interface. To do so you list the name of the interface after the name of the class and by using the implements keyword. In case of multiple interfaces simply separate them by comma after the implements keyword. For example:

    ```
    public class Employee implements Comparable, Cloneable{
         //class definition
    }
    ```

  - And You supply the definition for all the methods declared in all the interface for which you have declared your intention to implement
  - The reason for interfaces is that Java is a strongly typed language. When making a method call, the compiler needs to be able to check that the method actually exists. If an object belongs to a class that implements an interface then the compiler can be assured that the service that would invoke the interface method will be successful in doing so.
  - `Integer.compare(int a,int b)` or `Double.compare(double a,double b)` to compare two numbers. These methods return negative integer if a<b, zero if a=b and positive integer if a>b.

- **Properties of Interfaces**
  - Interfaces are not classes. In particular you can never create an object of / instantiate an interface by calling `new` on them.
  - You can although still declare a variable to be of interface type. An interface variable must refer to an object of a class that implements the interface of which type the variable is declared to be.
  - Just as you could use instanceof to check whether an object is an instance of a specific class, you can use it with an interface to check if an object belongs to a class that implements that interface.
  - Just as you can build hierarchies of classes, you can also extend interfaces. this allows for multiple chains of interfaces that go form a greater degree of generality to a greater degree of specialisation.
  - Interface as a concept was introduced to provide all the flexibility of multiple inheritance without the complexities that comes with it. So in Java, while each class can extend only one other class, they can implement multiple interfaces.

- If your class implements `Cloneable`, the clone methods in the `Object` class will make an exact copy of your class's objects.

- **Interfaces and Abstract Classes**
  - While interfaces and abstract classes feel similar, abstract classes can have instance fields and methods which operate on them while interfaces cant have any of these.
  - Other than this there is one major difference in the use of the two language constructs. A class can extend only one other class which can be a concrete or an abstract class but this means that a class can extend only one abstract class and once it extends an abstract class it won't be able to extend any other class. Interfaces on the other hand can be implemented by a class for as many interfaces that the class needs to implement/conform to.
  - So abstract classes can't be used to express a generic property with which if a class is tagged then it can avail certain services. But interfaces afford most of the benefits of multiple inheritance while avoiding the complexities and the inefficiencies. Interfaces hence can be used to express generic properties.
- **Static Methods**
  - From Java 8, You can have `static` methods now in interfaces
- **Default Methods**
  - You can supply a `default` implementation for any interface method. You must tag such a method with the `default` modifier. And the implementation cant use any instance fields—again because interfaces don't have any.
  - *But, a `default` **method** can call other methods of the interface in its implementation.*
  - If a programmer is happy with the default implementation of an interface method then he/she need not provide an implementation for the same and he can automatically use the default implementation.
  - An important use for default methods is interface evolution. Adding a new interface method to an interface would not be source compatible as other classes that have been implementing that interface upon being invoked with the new method will return `AbstractMethodError`.
- **Resolving Default Method Conflicts**
  - If the exact same method is defined as a default method in one of the interfaces implemented by a class and then again as a method in the superclass or another interface implemented by the class, it results in method conflicts.
  - **The rules to resolving method conflicts are:**

- Superclasses win. If a superclass provides a concrete method then the default methods from the implemented interfaces with same name and signature are simply ignored.
- Interfaces clash. If the method conflict arises because of two or more of the interfaces implemented by a class provide a default implementation, then the class facing the conflict needs to explicitly resolve the conflict.
  - While providing the explicit method definition in the class facing the conflict from interfaces it implements, simply specify the method that you want to keep.
- If even one interface provides a default implementation for a method that is declared with same method signature in another implementing interface as well, then the compiler reports an error and the programmer must explicitly define the method.

- These rules ensure that if you add default methods to an interface it has no effect on the code that worked before there were default methods.

- **Examples of Interfaces**

  - **Interfaces and Callbacks**
    - A common pattern in programming is the callback pattern. In this pattern, you specify the action that should occur whenever a particular event happens.
    - Example: When you construct a time object, you set the time interval and you tell it what it should do whenever the time interval has elapsed. You pass an object of some class. The timer then calls one of the methods of the object. Passing an object is more functional than passing a function because the object can carry additional information. The timer for this requires that you specify an object of a class that implements the `ActionListener` interface of the `java.awt.event` package. The timer calls the `actionPerformed` method when the time interval has expired.
  - **The `Comparator` Interface**
    - There is a second version of the `Arrays.sort` method whose parameters are an array and a comparator—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T>{
     int compare(T first, T second);
}
```

    - To compare strings by length, define a class that implements Comparator<String>.

```
class LengthComparator implements Comparator<String>{
     public int compare(String first, String second){
          return first.length-second.length;
}
```

Then to do the comparison you'll need to make an instance. The `compare` method is now called on the comparator object and not on the string:

```
        String[] friends = {“Mark”, “Bill”, “Elon”, ”Larry “};
        Arrays.sort(friends, new LengthComparator());
```

- **Object Cloning**
  - `Cloneable` is an interface that indicates that a class has provided a safe `clone` method.
  - When you make a copy of a variable holding an object reference, the original and the copy both refer to the same object which means that a change to either affects the other.
  - If you would like copy to be a new object that begins its life being identical to original but whose state can diverge overtime, use the `clone` method.
  - The clone method is a protected method of the `Object` class, which means that your code can’t just simply invoke that on an object. Only the class of the object whose copy you want can invoke clone on its objects.
  - There is an important reason for this restriction. The Object class knows nothing about the classes that will extend it, so it can only make a field by field copy of an object of one of those classes. If all the data fields in the object are numbers or primitive types, copying the fields works just fine. But if the object contains references to sub objects, then copying the fields gives you another reference to the same subject, so the original and the cloned objects still share some information. That is, the default cloning operation is “**shallow**”—it doesn't clone objects that are referenced inside other objects.
  - If the subject shared between the original and the shallow copy is immutable, then sharing is safe and we need not worry further. This happens if the subject belongs to an immutable class. Alternatively the subject may remain constant throughout the lifetime of either objects with no mutator accessing it and no methods yielding a reference to it.
  - However, in the case of subjects being mutable, you must redefine the `clone` method to make a **deep-copy** that clones the sub objects as well.
  - For every class you need to decide whether:
    - The default `clone` method is good enough/safe
    - The default `clone` method can be patched up by replacing mutable objects in the cloned object with their clones by calling `clone` on mutable subobjects and
    - if clone should not be attempted
  - The third option above is default. In order to follow any other option above, a class must:
    - Implement the `Cloneable` interface and
    - Redefine the clone method with public access modifier
  - In this case the appearance of the `Cloneable` interface has nothing to do with the normal use of interfaces. In particular, it does not specify the `clone` method—that method is inherited from the `Object` class. The interface merely serves as a tag,

indicating that the class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement the `Cloneable` interface.

> Even if the default (shallow copy) implementation of `clone` is adequate, you still need to implement the `Cloneable` interface, redefine the `clone` to be `public` and call `super.clone()`

```
public class Employee implements Cloneable{
     //raise the visibility level to public and change
     //the return type to the Class for which the clone
     //is being redefined
     public Employee clone() throws CloneNotSupportedException{
          return super.clone();
     }
}
```

- The `clone` method as shown above adds no special functionality to the shallow copy provided by `Object.class`. It merely makes the method `public`. At the same time, to make a deep copy, you'll have to clone the mutable instance fields.

```
public class Employee implements Cloneable{

     public Employee clone() throws CloneNotSupportedException{
          //call Object.clone()
          Employee other = (Employee) super.clone();
          //clone mutable fields
          other.hireDay = (Date)hireDay.clone();
          return cloned;
     }
}
```

- Whenever `clone` is invoked on an object whose class does not implement the `Cloneable` interface, the `clone` method of the `Object` class throws a `CloneNotSupportedException`
- It's usually a good option while designing the `clone` method to not catch the `CloneNotSupportedException` because it gives the subclasses the option to throw it if they can't support cloning. Catching it in case of `final` classes is fine.
- You have to be careful about cloning of subclasses. For example, once you have defined the clone method for the Employee class, anyone can use it to clone Manager objects. Can the Employee clone method do the job? It depends on the fields of the Manager class. In our case, there is no problem because the bonus field has primitive type. But Manager might have acquired fields that require a deep copy or are not cloneable. There is no guarantee that the implementor of the subclass has fixed clone to do the right thing. For that reason, the clone method is declared as protected in the Object class. But you don't have that luxury if you want users of your classes to invoke clone.

- All array types have a `clone` method that is public, not protected.

- **Lambda Expression**

  • **Why Lambdas**

  - ***<u>A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.</u>***

  - From examples in the previous section in Interfaces and callbacks or `Comparator` Interface, we effectively passed a block of code to someone—a timer or a `sort` method. This code block was called at some time later.

  - A simple example of a lambda expression would be:

```
//Implicit return statement
(String first, String second) ->
            first.length() - second.length()
```

    Such an expression is simply a block of code together with specification of any variables that must be passed to the code. This is one form of lambda expressions in Java: parameters, the `->` arrow, and an expression. If the code carries out a computation that doesn't fit in a single expression, then enclose the multiple statements in `{}` and with explicit `return` statement.

```
(String first, String second) ->
      {     //explicit return statements
            if (first.length()<second.length()) return -1;
            else if(first.length()>second.length()) return 1;
            else return 0;
      }
```

    It is illegal for a lambda expression to return a value in some branches but not in others.

  - If a lambda expression has no parameters, you still supply an empty parentheses, just like while defining a parameterless method.

```
() -> {for (int i=0; i<100; i++) System.out.println(i);}
```

  - If the parameter types of a lambda can be inferred, you can omit them. If a method has a single parameter with inferred type, you can even omit the parentheses. You never specify the result type of a lambda expression. It is always inferred from context. Example:

```
ActionListener listener = event ->
      System.out.println("The time is " + new Date();
```

  • **Functional Interfaces**

  - You can supply a lambda expression whenever an object of an *interface with a single abstract method* is expected. Such an interface is called a **functional interface**.

- To demonstrate the conversion to a functional interface, consider `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words, (first, second) ->
                    first.length()-second.length());
```

- Behind the scenes, the Arrays.sort method receives an object of some class that implements Comparator<String>. Invoking the compare method on that object executes the body of the lambda expression. the management of these objects and classes is completely implementation dependent, and it can be much more efficient than using a traditional inner class. It is best to think of a lambda expression as a function and not an object, and to accept that it can be passed to a functional interface.
- In fact, conversion to a functional interface is the only thing you can do with a lambda expression in Java.
- In Java API a number of very generic functional interfaces are defined in the `java.util.function` package. When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.
- A particularly useful interface in the java.util.function package is Predicate:

```
public interface Predicate<T> {
      boolean test(T t);
      // Additional default and static methods
}
```

  The `ArrayList` class has a `removeIf` method whose parameter is a `Predicate`. It is specifically designed to pass a lambda expression. For example, the following statement removes all `null` values from an array list:

```
list.removeIf(e -> e == null);
```

- **Method References**
  - Sometimes, there already is a method that carries out exactly the action that yo'd like to pass on to some other code. For example you want to pass the `println` method:

```
Timer t = new Timer(1000, event -> System.out.println(event));
//this can simply be written as
Timer t = new Timer(1000, System.out::println);
```

  - The expression `System.out::println` is a method reference that is equivalent to the lambda expression `x -> System.out.println(x);`
  - Another example: `Arrays.sort(strings, String::compareToIgnoreCase)`
  - As you can see from the above examples, the `::` operator separates the method name from the name of the object or class. There are 3 cases in which it can be used:

- `object::instanceMethod`
- `Class::staticMethod`
- `Class::instanceMethod`

- In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method.
- In the third case, the first parameter becomes the target (implicit parameter) of the method. For example `String::compareToIgnoreCase` is the same as `(x,y)->x.compareToIgnoreCase(y)`
- You can capture the this parameter in a method reference like `this::equals` which is the same as `x -> this.equals(x)`. It is also valid to use `super`. The method expression `super::instanceMethod` uses `this` as the target and invokes the superclass version of the given method.

• **Constructor References**

- Constructor references are just like method references, except that the name of the method is new. For example `Person::new` is a reference to a `Person` constructor. In case of multiple constructors with different signatures the choice of constructor is made based on the context of the constructor reference (what signature of arguments the constructor reference is called with).

```
ArrayList<String> names = . . .;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.collect(Collectors.toList());
```

- The `map` method, in the above example, calls the `Person(String)` constructor on each element of the list. In case of multiple `Person` constructors the compiler picks one which is called with a `String` based on the context in which the constructor is called.
- You can form constructor references with array types. For example int[]::new is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression x -> new int[x]. This is useful in getting a return array of correct type by passing the constructor reference to a method. For example:

```
//the stream interface has a toArray method
//that returns an Object array
Object[] people = stream.toArray();
//But passing a constructor reference to it helps return the
//correct array type
Person[] people = stream.toArray(Person::new);
```

• **Variable Scope**

- Often, you want to be able to access variables form an enclosing method or class in a lambda expression. A lambda expression has three ingredients:

- A block of code
- Parameter
- Values for the free variables, that is, the variables that are not parameters and not defined inside the code block and are rather accessed from the enclosing scope.

- The data structure representing the lambda expression must store the values for the free variables. Such values are said to have been captured by the lambda expression. The technical term for a block of code together with the values f the free variables is a closure.
- A lambda expression can capture the value of a variable in the enclosing scope. In Java, to ensure that the captured value is well-defined, a lambda expression can only reference variables whose values doesn't change. Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently. It is also illegal to refer to a variable in lambda expression that is mutated outside.

```
public static void countDown(int start, int delay, int count){
      for (int i=0;i<=count; i++){
            ActionListener listener = event -> {
                  start—;
                  //ERROR: Can’t mutate a captured variable
                  System.out.println(i);
                  //ERROR: Can’t reference a variable that changes
                  //outside
            }
      }
}
```

- The rule is that any captured variable in a lambda expression must be effectively final. An effectively final variable is a variable that is never assigned a new value after it has been initialised.
- The body of a lambda expression has the same scope as that of a nested block. The same rules for name conflicts and shadowing apps. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable. Inside a method you cant have two local variables with the same name and hence you cant introduce such variables in lambda expression either.
- When you use the `this` keyword in a lambda expression, you refer to the this parameter of the method that creates the lambda.
- The scope of the lambda expression is nested inside the method that creates is and so `this` keyword has the same meaning anywhere in that method.

- **Processing Lambda Expressions**
  - The point of using lambdas is deferred execution. After all, if you wanted to execute some code now, you'd do that without wrapping it inside a lambda.

- Reasons for executing code later:
    - Running the code in a separate thread
    - Running the code multiple times
    - Running the code at the right point in an algorithm
    - Running a code when something happens
    - Running the code only when necessary
- To accept a lambda in our methods, we need to pick (or in rare cases, provide) a functional interface. For common functional interfaces check out **page no. 326 Core Java Vol 1.**
- When working with primitive types as parameters for functional interface, it is a good idea to use specialisations the accept primitive types to reduce autoboxing. Checkout **Table 6.2 page 327 Core Java Vol 1.**

- **More about Comparators**
    - The `Comparator` interface has a number of convenient `static` methods for creating comparators. These methods are intended to be used with lambda expressions or method references.
    - The static comparing method takes a ***key extractor*** function that maps a type T to a comparable type. The function is applied to the objects to be compared, and then comparison is made on the returned keys. For example, when we want to sort people list based on `lastName` field of `Person` objects in it. You can chain comparators with the `thenComparing` method for breaking ties.:

```
Arrays.sort(people, Comparator.comparing(Person::getLastName)
          .thenComparing(Person::getFirstName));
//if two Person objects have same lastName then second
//comparator is used
```

    - There are a few variations of this method. You can even specify a comparator to be used for keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
     Arrays.sort(people, Comparator.comparing(Person::getName,
          (s,t) -> Integer.compare(s.length(),t.length())));
```

    - Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of int, long or double values. An easier way of producing the above operation is:

```
     Arrays.sort(people,
          Comparator.comparingInt(p -> p.getName().length()));
```

    - If you key function can return null, you should use the nullsFirst and nullsLast adapters. These static methods take an existing comparator and modify it so that it doesn't throw an exception when encountering null values but ranks them as smaller or larger than regular values. The nullsFirst method needs a comparator—in this case, one that compares two strings. The naturalOrder method makes a comparator for any class implementing `Comparable`. `Comparator.<String>naturalOrder()` is what we need here. The type for `naturalOrder` is inferred.

```
import static java.util.Comparator.*

Arrays.sort(people, comparing(Person::getMiddleName,
                              nullsFirst(naturalOrder())));
```

- The static `reverseOrder` method gives the reverse of the `naturalOrder`. To reverse any comparator, use the `reversed` instance method. For example, `naturalOrder().reversed()` is the same as `reverseOrder()`.

- **Inner Classes**
  - An inner class is a class that is defined inside another class. There are three reasons for doing so:
    - Inner class methods can access the data from the scope in which they are defined—including the data that would otherwise be private.
    - Inner classes can be hidden from other classes in the same package.
    - Anonymous inner classes are handy when you want to define callbacks without writing a lot of code.

- **Use of an Inner Class to Access Object State**
  - An inner class method gets access to both its own data fields and those of the outer object creating it.

```
public class TalkingClock{
      private int interval;
      private boolean beep;

      public TalkingClock(int interval, boolean beep){
            this.interval = interval;
            this.beep = beep;
      }

      public void start(){.. .. .. }

      public class TimePrinter implements ActionListener{
            //an inner class
            public void actionPerformed(ActionEvent event){
                  if (beep) Toolkit.getDefaultToolkit().beep();
            }
      }
}
```

- For this to work, an object of an inner class always gets an implicit reference to the object that created it. The outer class reference is set in the constructor. The compiler modifies all inner class constructors, adding a parameter for the outer class reference. Since the TimePrinter class defines no constructors in the example above, the complier synthesises a no-argument constructor and sets the reference to outer object in that.
- There is no need to provide accessors that are of interest to only one other class. If possible make that other class an inner class of the class whose fields it is interested in.

• **Special Syntax Rules for Inner Classes**

- The expression `OuterClass.this` denotes the outer class reference. So to be more explicit you could write the actionPerformed method above as:

```
public class TimePrinter implements ActionListener{
      //an inner class
      public void actionPerformed(ActionEvent event){
            if (TalkingClock.this.beep)
                  Toolkit.getDefaultToolkit().beep();
      }
}
```

- Conversely you can write the call to inner class object constructor as :

```
      outerObject.new InnerClass(constructor parameters)

      //or in our example
      ActionListener listener = this.new TimePrinter();
```

- Here the outer class reference of the newly constructed TimePrinter object is set to the `this` reference of the method that creates the inner class object. As always this. qualifier is redundant. However it is also possible to set the outer class reference to another object by explicitly naming it.
- You can refer to an inner class as `OuterClass.InnerClass` when it occurs outside the scope of the outer class.

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

- Any `static` fields declared in an inner class must be `final`. One expects a unique instance of a static field, but there is a separate instance of the inner class for each outer class object. An inner class cannot have `static` methods.

• **Are Inner Classes Useful? Actually Necessary?Secure?**

- Inner classes are a phenomenon of the *compiler*, not the virtual machine. Inner classes are translated into regular class files with $ (dollar signs) delimiting outer and inner class names, and the virtual machine does not have any special knowledge about them.
- `javap` utility prints declarations of the non-private members of a class. Running:

```
javap -private innerClass.TalkingClock\$TimePrinter
```

You will get the following printout:

```
public class TalkingClock$TimePrinter{

      public TalkingClock$TimePrinter(TalkingClock);

      public void actionPerformed(java.awt.event.ActionEvent);

      final TalkingClock this$0;
}
```

- As you can see above, the compiler has added an instance field `this$0` for reference to the outer class. As well the parameter for outer class has been added in the inner class constructor.
- Inner classes are genuinely more powerful than regular classes because they have more access privileges. While an inner class can access private members of an outer class, a regular class with a reference to the outer class won't be able to do the same.
- The compiler adds a new method to the class with a cryptic name and it is this class that the inner class calls to get the value of private outer class field. So If an inner class can access a private data field, then it is possible to access that data field through classes added to the package of the outer class but that requires skill and determination.
- The synthesised constructors and methods can get quite convoluted. Suppose we turn `TimePrinter` into a `private` inner class. There are no `private` classes in the virtual machine, so the compiler produces the next best thing: a package-visible class with a `private` constructor:

  ```
  private TalkingClock$TimePrinter(TalkingClock);
  ```

  Of course, nobody can call that constructor, so there is a second package-visible constructor:

  ```
  TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
  ```

  that calls the first one. The `TalkingClock$1` class is synthesised solely to distinguish this constructor from others. The compiler translates the constructor call in the start method of the `TalkingClock` class to

  ```
  new TalkingClock$TimePrinter(this, null)
  ```

- **Local Inner Classes**
  - If you look in the TalkingClock example above, you'll find that you need the name of the type TimePrinter only once in the start method when you create an object of that type. In a situation like this, you can simply define the class locally inside a single method.

- Local classes are never defined with an access specifier(that is public or private). Their scope is always restricted to the block in which they are declared.
- They are completely hidden from the outside world—not even other code in the TalkingClock class can access them. No method except start has any knowledge of it.

```
public void start() {

      //No access modifier
      class TimePrinter implements ActionListener {
            public void actionPerformed(ActionEvent event) {
                  System.out.println("The time is " + new Date());
                  if (beep) Toolkit.getDefaultToolkit().beep();
            }
      }

      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener); t.start();
}
```

- **Accessing Variables from Outer methods**
  - Local classes have one advantage over inner classes: not only can they access the instance fields of the outer class but they can also access the local variables of the method. However those variables have to be effectively final. That means they may never change once they have been assigned.
  - Consider the following example:

```
   public void start(int interval, boolean beep) {
      class TimePrinter implements ActionListener {
            public void actionPerformed(ActionEvent event) {
                  System.out.println("The time is " + new Date());
                  if (beep) Toolkit.getDefaultToolkit().beep();

            }
      }

      ActionListener listener = new TimePrinter(); Timer t = new
      Timer(interval, listener); t.start();
   }
```

  - In the above example:
    1. The start method is called.
    2. The object variable listener is initialised by a call to the constructor of the inner class TimePrinter.

3. The listener reference is passed to the Timer constructor, the timer is started, and the start method exits. At this point, the beep parameter variable of the start method no longer exists.

4. A second later, the actionPerformed method executes if (beep) . . .

- For the code in the `actionPerformed` method to work, the `TimePrinter` class copies the `beep` field as a local variable of the `start` method, before the beep parameter value went away. In our example, the compiler synthesises the a variable `final boolean val$beep` for the `beep` method parameter and `TalkingClock$1TimePrinter` name for the local inner class.

- The compiler detects access of local variables, makes matching instance fields for each one, and copies the local variables into the constructor so that the instance fields can be initialised. When an object is created, the value `beep` is passed into the constructor and stored in the `val$beep` field. The methods of a local class can refer only to local variables that are declared final. For more check out **Page 341 Core Java Vol 1**

- A local variable that is declared final cannot be modified after it has been initialised. Thus, it is guaranteed that the local variable and the copy made inside the local class will always have the same value

- If you need to update the value of a local or instance variable in the enclosing scope of an inner class, use an array of that type and update its value. But remember that, in case of concurrent programming, such a situation is prone to race condition.

• **Anonymous Inner Classes**

- If you want to make only a single object of an inner class, then you don't even need to give the class a name. Such a class is called an anonymous inner class.

```
public void start(int interval, boolean beep){

     ActionListener listener = new ActionListener(){
          public void actionPerformed(ActionEvent event){
               System.out.println("Time : " + new Date());
               if (beep) Toolkit.getDefaultToolkit().beep();
          }
     };

     Timer t = new Timer(interval, listener);
     t.start();
}
```

- What the syntax above means is: Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces.

- In general, the syntax is:

```
new SuperType(construction parameters){
     // inner class methods and data
}
```

- Here, the SuperType can be an interface, in which case the inner class implements that interface, or it can also be a class, in which case the inner class extends that class.
- An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of the class, and an anonymous class has no names. Instead the construction parameters are given to eat superclass constructors. If the closing parentheses of the construction parameter list is followed by an opening brace, then an anonymous inner class is being defined.
- In particular, whenever an inner class implements an interface, it cannot have any construction parameters. Nevertheless, you must supply a set of parentheses as in:

```
Person queen = new Person("Mary");
// a Person object

Person count = new Person("Dracula") { . . . };
// an object of an inner class extending Person

new InterfaceType(){
     //methods and data
}
```

- If you don't need an array list again after its first use, you can make it anonymously and pass it to whichever method you need to pass it to.

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Tony");
invite(friends);

//becomes
invite(new ArrayList<String>(){{add("Harry"); add("Tony");}});
```

- You need to be careful with `equals` method and objects of an anonymous subclass as it would fail the `getClass() != other.getClass()` test.
- When logging you may want the name of the current class. getClass() fails in a static method as it calls `this.getClass()`, and a `static` method has no `this`. Use the following expression instead:

```
new Object(){}.getClass().getEnclosingClass()
// gets class of static method
```

Here, `new Object(){}` makes an anonymous object of an anonymous subclass of Object, and getEnclosingClass gets its enclosing class—that is, the class containing the static method.

- **Static Inner Classes**
  - Occasionally you may want to use an inner class simply to hide one class inside another—but you don't need the inner class to have a reference outer class object. You can suppress the generation of that reference by declaring the inner class `static`.
  - Only inner classes can be declared `static`. A `static` inner class is exactly like any other inner class except that an object of `static` inner class does not have a reference to the outer class that generated it.
  - In case when an inner class object is constructed inside a `static` method its corresponding inner class also needs to be/ must be `static`. Use a static inner class whenever the inner class does not need to access an outer class object.
  - Unlike regular inner classes, static inner classes can have static fields and methods. Inner classes that are declared inside an interface are automatically `static` and `public`.

```
class ArrayMinMaxAlgorithm {

    //static inner class
    public static class Pair {
        //method code here
    }

    //class code here

    public static Pair minmax(double[] d) {
        ...

        return new Pair(min, max);
     }
}
```

- **Proxies**
  - This is given in a really good and concise manner in the book. Read **Page 350 onwards Core Java Vol 1.**

- **Exceptions, Assertions, and Logging**

- **Dealing with Errors**
  - Users expect that the programs will act sensibly when errors happen. If an operation cannot be completed because of an error, the program ought to either:
    - Return to a safe state and enable the user to execute other commands
    - Allow the user to save all work and terminate the program gracefully.

- This may not be easy to do, because the code that detects or causes the error condition is usually far removed from the code that can roll back the data to a safe state or to the code that can save user's work and exit gracefully.
- The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation. To handle exceptional situations in your program, you must take into account the errors and problems that may occur.
  - User input errors i.e incorrect input
  - Device errors i.e hardware doesn't always do what you want it to
  - Physical limitations i.e disk can fill up or you can run out of available memory
  - Code errors i.e. a method may not perform correctly.
- The traditional reaction to an error in a method is to return a special error code that the calling method analyses. But there may not be an obvious way of distinguishing between valid and invalid data always so specific return value is not feasible.
- In case a method is not able to complete its task in the normal way, Java allows methods to throw an object that encapsulates the error information. In this case, the method exits immediately, it does not return any value and the execution does not resume at the code that called the method; instead the exception handling mechanism begins its search for an exception handler that can deal with this particular error.
- Exceptions have their own syntax and are part of a special inheritance hierarchy.

• **The Classification of Exceptions**

- An exception object is always an instance of a class derived from `Throwable` i.e. all exceptions descend from it but the hierarchy splits immediately into two branches `Error` and `Exception`.
- The `Error` hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system. You should not throw an object of this type. There is little you can do if such an error occurs beyond notifying the users and trying to terminate the program gracefully.
- While programming in Java, its the `Exception` hierarchy objects that we throw. The Exception hierarchy splits into `RuntimeException` and `IOException`.
- A `RuntimeException` happens because you made a programming error. Any other exception occurs because a bad thing, such as I/O error happened to your otherwise good program.
- Exceptions that inherit form `RuntimeException` include such problems as:
  - A bad cast
  - An out of bounds array access

    - A null pinter access
- Exceptions that inherit form `IOException` include:
    - Trying to read past the end of a file
    - Trying to open a file that doesn't exist
    - Trying to find a `Class` object form a string that does not denote an existing class.
- The rule is : "**If it is a RuntimeException, it was your fault".**
- The Java Language specification calls any exception that derives from the class `Error` or the class `RuntimeException` an **unchecked exception**.
- All other exceptions are called **checked exceptions**. The compiler checks that you provided exception handlers for all checked exceptions.

- **Declaring Checked Exceptions**
    - A Java method can throw an exception if it encounters a situation it cannot handle. A method will not only tell the Java Complier what values it can return, it is also going to tell the Java compiler what can go wrong.
    - For example: code that attempts to read from a file knows that the file might not exist or that it might be empty. The code that tries to process the information in a file therefore will need to notify the compiler that it can throw some sort of `IOException`.
    - You advertise that your method can throw an exception in the header of the method i.e. the header changes to reflect the checked exceptions that a method can throw.

```
public Image loadImage(String imageName) throws IOException{
      //method code goes here
}
```

```
public FileInputStream(String name) throws FileNotFoundException{
      //constructor code goes here
}
```

- The declarations above say that the method `loadImage` can throw an `IOException` whereas the constructor `FileInputStream` can throw `FileNotFoundException`.
- When you write your own methods you don't have to mention every throwable object that your method might throw. Only when:
    - You call a method in you method that throws a checked exception
    - You detect an error and throw a checked exception with the throw statement
- Only in the above two cases should you declare the exceptions in the method declarations so that other programmers who use your method know about the possibility if an exception. Then if they don't provide any handler to catch the exception, the thread of execution terminates.

- You declare that your method may throw an exception with an exception specification using the `throws` keyword (followed by the list of classes of the checked exceptions that might be thrown during the execution of your method) in the method header.
- You do not need to advertise internal Java errors—that is exceptions inheriting from `Error`. Any code could potentially throw those exceptions and they are entirely beyond your control.
- Similarly, you should not advertise unchecked exceptions inheriting from `RuntimeException`. The runtime errors are completely under your control.
- In summary, a method must declare all the checked exceptions that it might throw. Unchecked exceptions are either beyond your control (`Error`) or result from conditions that you should not have allowed (`RuntimeException`). If your methods fails to declare all the checked exceptions faithfully, the compiler will issue an error message.
- Also instead of declaring the exception, you can catch and handle it and then you wont have to specify it in `throws`.
- If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method. (It is OK to throw more specific exceptions, or not to throw any exceptions in the subclass method.) In particular, **if the superclass method throws no checked exception at all, neither can the subclass**
- When a method in a class declares that it throws an exception that is an instance of a particular class, it may throw an exception of that class or any of its subclasses.

• **How to Throw an Exception**

- To throw an exception you decide on what exception best describes what went wrong and then simply create a new instance of that exception and throw is as following (eg for `EOFException`):

```
throw new EOFException();
```

- Here's how it all fits together:

```
String readData(Scanner in) throws EOFException{
      .. .. ..
      while(.. .. ..){
            if (in.hasNext()){    //EOF encountered
                  if(numWords < lengthDeclared){
                        throw new EOFException();
                  }
            .. .. ..
            }
      }
      return s;
}
```

- The EOFException has a second constructor that takes a string argument. You can pu this to good use by describing the exceptional condition more carefully.
- Throwing an exception is easy if one of the existing exception classes work for you. You simply:
  - Find an appropriate exception class
  - Make an object of that class
  - Throw it.

- **Creating Exception Classes**
  - Your code may run into a problem which is not adequately described by any of the standard exception classes.
  - In this case, it is easy enough to create your own exception class. Just derive it from `Exception`, or from a child class of `Exception` such as `IOException`. It is customary to give both a default constructor and a constructor that contains a detailed message.

```
public class FileFormatException extends IOException{
      public FileFormatException(){}
      public FileFormatException(String message){
            super(message);
      }
}
```

  - After this you can simply import this class whenever you need to throw an exception instance of this class, create the instance and throw it where such an exception condition is met.

- **Catching Exceptions**
  - **Catching an Exception**
    - If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace.
    - To catch an exception, set up a `try/catch` block as follows:

```
try{
      //code with potential to throw
      //Exception goes here
}
catch (ExceptionType1 e1){
      //handler code for this exception type goes here
}
```

```
        catch (ExceptionType2 e2){
              //handler code for this exception type goes here
        }
```

- If any code inside the try block throws an exception of the class specified in one of the catch clauses:
    - The program skips the remainder of the code in the `try` block
    - The JVM matches the most specific exception type of the types in the `catch` clauses that the thrown exception object is an instance of and then executes the handler code inside its `catch` clause.
- If any code in the method throws an exception of type other than the one named in the `catch` clause, this method exits immediately.
- Often the best choice while dealing with code that might throw an exception is to do nothing and simply pass the exception on to the caller of the method and let it worry about it. If we follow this approach then, we'd have to declare the exception in the method header.
- The compiler strictly enforces the throws specifiers. If you call a method that throws a checked exception, you must either handle it or pass it on.
- We have already seen how to catch an exception but if you decide to pass on an exception then in that case you need no make any special changes in the code and simply declare the exception in the method header.
- As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle. It is better to direct an exception to a competent handler than to squelch it.
- If you are writing a method that overrides a superclass method which throws no exceptions then you must catch each checked exception in the your methods code. *You are not allowed to add more throws specifier to a subclass method than are present in the superclass method.*

• **Catching Multiple Exceptions**

- You can catch multiple exception types in a `try` block and handle each type differently. Use a separate `catch` clause for each type.
- The exception object may contain information about the nature of the exception. To find out more about the object try:

```
        e.getMessage() //to get detailed error message (if set) or
        e.getClass().getName()
```

  to get the actual type of the exception object.

- As of Java SE7, if handler code for multiple exception types is common then you can catch multiple exception types in the same catch clause. You can **combine the exceptions using** " | "**(single pipe symbol)** as follows:

```
try{
     //code that might throw exception
}
catch (FileNotFoundException | UnknownHostException e){
     //emergency action for missing field and unknown hosts
}
catch (IOException e){
     //emergency action for all other I/O problems
}
```

- This is only needed when catching exception types that are not subclasses of one another.

• **Rethrowing and Chaining Exceptions**

- **You can throw an exception in a `catch` clause**. Typically you do this when you want to change the exception type. If you build a subsystem that other programmers use, it makes a lot of sense to use an exception type that indicates a failure of the subsystem. Example: `ServletException`. The code that executes a servlet may not want to know in minute detail what went wrong but it definitely needs to know that the servlet was at fault.
- However while throwing a new exception, it is a better idea to set the original exception as the cause of the new exception.

```
try{
     //access the database
}
catch (SQLException e){

     Throwable se = new ServletException("Database Error");
     //set the init cause as the exception
     se.initCause(e);
     throw se;
}
```

- This wrapping technique is highly recommended. It allows higher level exceptions in subsystems without losing the details of the original failure. When the exception is caught, the original exception can be retrieved as :

```
Throwable e = se.getCause();
```

The wrapping technique is also useful if a checked exception occurs in a method that is not allowed to throw a checked exception. You can catch the checked exception and wrap it into a runtime exception. (Bad practice result of bad design though)

- Sometimes you may just wanna log an exception and rethrow it without any change. More about logging later.
- In the above example, the compiler tracks the fact that `se` originates from the `try` block. Provided that the only checked exceptions in that block are `ServletException` instances, and provided that e is not changed in the `catch` block, it is valid to declare the enclosing method as `throws ServletException`.

• **The `finally` Clause**

- When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up. For example, it is very important to close all database connections properly, even when exceptions occur. Use `finally` clause for such cases. The code in the `finally` clause executes whether or not an exception was caught.

```
InputStream in = new FileInputStream(. . .);
try{
      //code that might throw exception
}
catch(IOException e){
      //handler for IOException
}
finally{
      in.close() //gets called no matter what
}
```

- Just to update the control flow:
  - If no exception occurs then all the lines in the `try` block are executed and then control moves to `finally` block. After executing all lines in the `finally` block, the control then moves to the line after the `finally` block.
  - If an exception is thrown, then the rest of the lines in the `try` block are skipped and control moves to handler code in the appropriate `catch` block.
    - If no exception is thrown in the `catch` clause execution, then the execution moves to `finally` block and then to the line after `finally` block.
    - Else, the exception is thrown back to the caller of the method.
  - If exception occurs and no `catch` clause fits the exception thrown then the rest of the `try` block is skipped, and `finally` block is executed. After this the exception is thrown back to the caller of the method.
- You can use the `finally` clause without the `catch` clause. In such a case the finally clause is executed whether or not an exception is thrown. If an exception is thrown, it is rethrown but the finally block ensures that resources are closed in the end no matter what.

- **[TIP]**: We strongly suggest that you ***decouple*** `try/catch` and `try/finally` blocks. This makes your code far less confusing. For example:

```
InputStream in = . . .;
try{
      try{
            //code that might throw exceptions
      }
      finally{
            in.close();
      }
}
catch (IOException e) {
      //show error message
}
```

  The inner try block has a single responsibility: to make sure that the input stream is closed. The outer try block has a single responsibility: to ensure that errors are reported. Not only is this solution clearer, it is also more functional: Errors in the finally clause are reported

- **[CAUTION]:** A `finally` clause can yield unexpected results when it contains return statements. Suppose no exception is thrown and you exit the try block with a `return` statement. Before the method returns, the `finally` block is executed. If the `finally` block also contains a `return` statement, then it masks the original return value and the `finally` return value is returned instead.

- If an exception is thrown in the execution of the `finally` block then the original exception that was meant to be thrown is lost the the exception thrown in the `finally` block is thrown instead. One way of avoiding it is as below:

```
InputStream in = . . .;
Exception ex = null;
try{
      try {
            //code that might throw exceptions
      }
      catch (Exception e) {
            ex=e;
            throw e;
      }
}
finally {
      try {
            in.close();
      }
      catch (Exception e) {
            if (ex == null) throw e;
```

```
                }
        }
```

- **The Try-with-Resources Statement**
    - Since Java SE7, if a resource belongs to a class that implements the `AutoCloseable` interface (with a single interface method: `void close() throws Exception`), then you can use the try-with-resource statement as below:

```
        try(ResourceClass1 res1 = . . .;
            ResourceClass2 res2 = . . .)
        {
              //use res1 and res2 in code
              //this block may throw exception
        }
```

      When the `try` block exits (either due to exception or normal processing completed), then the `res1.close()` and `res2.close()` are called automatically. You can specify multiple resources by separating the statements with a semi-colon as shown above.

    - Using this approach, the original exception is rethrown and any exceptions thrown by the close methods of the resources are considered suppressed. They are automatically caught and added to the original exception with the `addSuppressed` method. If you are interested in them, call the `getSuppressed` method which yields an array of the suppressed expressions from `close` methods.
    - **Use the try-with-resources statement whenever you need to close a resource**

- **Analysing Stack Trace Elements**
    - A ***stack trace*** is a listing of all pending method calls at a particular point in the execution of a program. They are displayed whenever a Java program terminates with an uncaught exception.
    - You can access the text description of a stack trace by calling the `printStackTrace` method of the `Throwable` class. A more flexible approach is the `getStackTrace` method that yields an array of `StackTraceElement` objects, which you can analyse in your program.

```
    Throwable t = new Throwable();

    t.printStackTrace();     //to print stack trace to console

    //to analyse stack frames
    StackTraceElement[] frames = t.getStackTrace();
    for (StackTraceElement frame : frames){
        //analyse frame
    }
```

The `StackTraceElement` class has methods to obtain the file name and line number, as well as the class and method name, of the executing line of code. The `toString` method yields a formatted string containing all of this information.

- The static `Thread.getAllStackTraces` method yields the stack traces of all threads. Here's how to use it:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet()){
   StackTraceElement[] frames = map.get(t);
   //analyse frames
}
```

- Check Throwable, Exception, StackTraceElement API on **Page 379-381 Core Java Vol 1**

- Tips for Using Exceptions

• **Exception handling is not supposed to replace a simple test (condition check to see if you might run into exception and avoiding it then)**. If a simple test can help like checking if a stack is empty before popping rather than catching EmptyStackException later, write conditional check. **Use exceptions for exceptional circumstances only**.

• **Do not micromanage exceptions.** Don't wrap each exception-potent statement in a separate try block. In case of multiple statements that might throw exceptions, put them in a block together. For example, the code looks much cleaner. It fulfils one of the promises of exception handling: to ***separate*** **normal processing from error handling**.

```
try {
   //only separate error handling from normal processing
   //no need to wrap each exception-potent statement in a
   //separate try block
   for (i = 0; i < 100; i++) {
         n = stack.pop();
         out.writeInt(n);
   }

}
catch (IOException e) {
   // problem writing to file
}
catch (EmptyStackException e) {
   // stack was empty
}
```

- **Make good use of the exception hierarchy**. Find an appropriate subclass of Exception or create your own custom exception to better reflect the exceptional circumstance encountered. Don't just catch Throwable. It makes your code hard to read and maintain. Respect the difference between checked and unchecked exceptions. Checked exceptions are inherently burdensome—don't throw them for logic errors. Do not hesitate to turn an exception into another exception that is more appropriate.
- **Do not squelch exceptions**. Even if an exception happens rarely, if you believe that exceptions are at all important, you should make some effort to handle them right and not not catch and ignore them.
- **When you detect an error, "tough love" works better than indulgence.** It is better to throw an exception if you can't handle it properly so that another competent code can handle it right.
- **Propagating exceptions is not a sign of shame.** Many programmers feel compelled to catch all exceptions that are thrown. Often, it is actually better to *propagate* the exception instead of catching it. **Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.**

- Using Assertions

- **The Assertion Concept**
  - The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.
  - The Java language has a keyword assert. There are two forms:

    ```
    assert condition;
    and
    assert condition : expression;
    ```

    Both statements evaluate the condition and throw an `AssertionError` if it is false. In the second statement, the expression is passed to the constructor of the `AssertionError` object and turned into a message string.
  - The sole purpose of the *expression* part is to produce a message string. The `AssertionError` object does not store the actual expression value, so you can't query it later.
  - For example, To assert that x is non-negative, you can simply use the statement:

    ```
    assert x >= 0;
    ```

    Or you can pass the actual value of x into the `AssertionError` object, so that it gets displayed later.

```
        assert x >= 0 : x;
```

- **Assertion Enabling and Disabling**
  - By default, assertions are disabled. Enable them by running the program with the `-ea` or `-enableassertions` option:

    ```
    java -enableassertions MyApp
    ```

  - You do not have to recompile your program to enable or disable assertions. Enabling or disabling assertions is a function of the *class loader*. When assertions are disabled, the class loader strips out the assertion code so that it won't slow execution. You can even turn on assertions in specific classes or in entire packages. For example:

```
java -ea:MyClass -ea:com.mycompany.mylib... -da:MyOtherClass MyApp
```

  This command turns on assertions for the class `MyClass` and all classes in the `com.mycompany.mylib` package *and its sub-packages*. The option `-ea...` turns on assertions in all classes of the default package. You can also disable assertions in certain classes and packages with the `-da` or `-disableassertions`:

  - Some classes are not loaded by a class loader but directly by the virtual machine. You can use these switches to selectively enable or disable assertions in those classes. However, the `-ea` and `-da` switches that enable or disable all assertions do not apply to the "system classes" without class loaders. Use the `-enablesystemassertions or -esa` switch to enable assertions in system classes.

- **Using Assertions for Parameter Checking**
  - The Java language gives you three mechanisms to deal with system failures:
    - Throwing an exception
    - Logging
    - Using Assertions
  - While deciding to use Assertions, keep in mind that:
    - Assertion failures are intended to be fatal, unrecoverable errors.
    - Assertion checks are turned on only during development and testing
  - Therefore, you would not use assertions for signalling recoverable conditions to another part of the program or for communicating problems to the program user. Assertions should only be used to locate internal program errors during testing.
- **Using Assertions for Documenting Assumptions**

- Assertions are a tactical tool for testing and debugging. In contrast, logging is a strategic tool for the entire lifecycle of a program. The example below shows how one might document assumptions using assertions in the code.

```
assert i>=0;
if(i%3==0) ...
else if (i % 3 == 1) ...
else {
assert i % 3 == 2;
... }
```

- **Logging**
  - **To be continued Later**
    -

- **Generic Programming**
  - Generic classes are desirable because they let you write code that is safer and easier to read than code littered with `Object` variables and casts. Generics are particularly useful for collection classes, such as the ubiquitous `ArrayList`.

- **Why Generic Programming?**
  - *Generic programming* means writing code that can be reused for objects of many different types

- **The Advantage of Type Parameters**
  - Before generic classes were added to Java, generic programming was achieved with *inheritance.* This approach has two problems:
    - A cast is necessary whenever you retrieve a value
    - Moreover, there is no error checking. You can add values of any subclass and the code would compile perfectly. It is elsewhere that when you retrieve the object and apply a cast that you will get an error.

- Generics offer a bester solution, type parameters that indicates the element type in a being handled in particular instance of that generic class. This makes the code easier to read. The compiler can make good use of the type information too. The compiler knows the type of the value retrieved hence no casts is required. The compiler also knows the type of the parameter of methods of on a particular instance of generic class. This is a lot safer than having an `Object` parameter. Now the compiler can check that you don't insert objects of the wrong type. A compiler error is much better than a `ClassCastException` at runtime.

- **Who Wants to be a Generic Programmer**
  - It is not very easy to implement a generic class. The programmers who use your code will want to plug in all sorts of classes for your type parameters. They will expect everything to work without onerous restrictions and confusing error messages. Your job as a generic programmer, therefore, is to anticipate all the potential future uses of your class.
  - Wildcard types are rather abstract, but they allow a library builder to make methods as flexible as possible.
  - Generic programming falls into three skill levels.
    - At a basic level, you just use generic classes—typically, collections such as ArrayList—without thinking how and why they work. Most application programmers will want to stay at that level until something goes wrong. You may encounter a confusing error message when mixing different generic classes, or when interfacing with legacy code that knows nothing about type parameters;
    - At that point, you need to learn enough about Java generics to solve problems systematically rather than through random tinkering.
    - Finally, of course, you may want to implement your own generic classes and methods.

- **Defining a Simple Generic Class**
  - A generic class is a class with one or more type variables. To introduce type variables while defining a generic class, **<u>use angle brackets after the class name and place type parameters in it separated by commas.</u>**
  - The type parameters are then used throughout the class definition to specify method return types and the types of fields and local variables. For example:

```
public class SampleGenericClass<T, U> {
     private T first;
     private U second;
```

```
        public SampleGenericClass() {
            this.first = null;
            this.second = null;
        }

        public SampleGenericClass(T first, U second) {
            this.first = first;
            this.second = second;
        }

        public T getFirst() {
            return first;
        }

        public U getSecond() {
            return second;
        }

        public void setFirst(T newValue){
            this.first = newValue;
        }

        public void setSecond(U newValue){
            second = newValue;
        }
    }
```

- The official Sun/Oracle Java naming convention for generic type parameters is:
    - *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 instantiate the generic type by substituting types for the type variables**, such as:

```
    SampleGenericClass<String>
```

  You can think of the result as an ordinary class with constructors:

```
        SampleGenericClass<String>()
        SampleGenericClass<String>(String, String)
```

and methods:

```
        String getFirst()
        String getSecond()
        void setFirst(String)
        void setSecond(String)
```

In other words, the generic class acts as a factory for ordinary classes.

- **Generic Methods**
  - In the preceding section, you saw how to define a generic class with compatible type parameter methods inside that generic class.
  - However, You can define generic methods both inside ordinary classes and inside generic classes. To do so in ordinary classes, t**<u>he type variables are inserted in angle brackets after the modifiers (i.e `public static` etc) and before the return type</u>**. Example:

```
    public class ArrayAlgorithmClass {

        public static <T> T getMiddle(T... a) {
            // ... in parameters after parameter type means the
            // method takes variable number of arguments of
            // the specified type

            return a[a.length / 2];
        }

        public <T> int defaultGetHashCode(T sampleParameter){
            return sampleParameter.hashCode()
        }

    }
```

  - **<u>When you call a generic method, you can place the actual types, enclosed in angle brackets, before the method name</u>**:

```
String middle =
    ArrayAlgorithmClass.<String>getMiddle("John", "Q.", "Public");
```

  - In most cases, you can omit placing the type parameter like `<String>` (as given in the above example) from the method call. The compiler has enough information to infer the method that you want to call. In almost all cases, type inference for generic methods works smoothly. Occasion- ally, the compiler gets it wrong, and you'll need to decipher an error report.

Tuesday, 7 March 2017

- **Bounds for Type Variables**
    - Sometimes, a class or a method needs to place restrictions on type variables. For example:

```
class ArrayAlgorithmClass {

   public static <T> T min(T[] a){ // almost correct

         if (a == null || a.length == 0)
              return null;
         T smallest = a[0];
         for (int i = 1; i < a.length; i++){
              if (smallest.compareTo(a[i]) > 0){
                   smallest = a[i];
              }
         }
         return smallest;
   }
}
```

    - So in the above example, everything looks fine but there is a problem. The variable `smallest` has type T, which means that it could be an object of an arbitrary class. H*ow do we know that the class to which T belongs has a compareTo method*?
    - The solution is to restrict `T` to a class that implements the `Comparable` interface—a standard interface with a single method, `compareTo`. You can achieve this by giving a *bound* for the type variable `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

      Now, the generic `min` method can only be called with arrays of classes that implement the `Comparable` interface . Calling this method with an array of any other class that does not implement `Comparable` is a compile time error.

    - The notation in specifying bound in the above example:

```
<T extends BoundingType>
```

      expresses that T should be a *subtype* of the bounding type. Both T and the bounding type can be either a class or an interface. The `extends` keyword was chosen because it is a reasonable approximation of the subtype concept, and the Java designers did not want to add a new keyword (such as sub) to the language.

    - A type variable or wildcard can have multiple bounds. The bounding types are separated by ampersands (&) because commas are used to separate type variables.

```
<T extends Comparable & Serializable>
```

    - **As with Java inheritance, you can have as many interface supertypes as you like, but at most one of the bounds can be a class. If you have a class as a bound, it must be the first one in the bounds list.**

- **Generic Code and the Virtual Machine**
  - The virtual machine does not have objects of generic types—all objects belong to ordinary classes
  - **Type Erasure**
    - Whenever you define a generic type, a corresponding *raw* type is automatically provided. The name of the raw type is simply the name of the generic type, with the type parameters removed. The type variables are *erased* and replaced by their bounding types (or `Object` for variables without bounds).
    - Our example from above becomes:

```
public class SampleGenericClass {
   private Object first;
   private Object second;

   public SampleGenericClass() {
        this.first = null;
        this.second = null;
                                }

   public SampleGenericClass(Object first, Object second) {
        this.first = first;
        this.second = second;
   }

   public Object getFirst() {
        return first;
   }

   public Object getSecond() {
        return second;
   }

   public void setFirst(Object newValue){
        this.first = newValue;
                           }

   public void setSecond(Object newValue){
        second = newValue;
   }
}
```

    - Since `T` was an unbounded type variable, it is simply replaced by `Object`. The result is an ordinary class, just as you might have implemented it before generics were added to Java.

- Your programs may contain different kinds of `SampleGenericClass`, such as `SampleGenericClass<String>` or `SampleGenericClass<LocalDate>`, but erasure turns them all into raw `SampleGenericClass` types.
- **The raw type replaces type variables with the first bound, or `Object` if no bounds are given.** Since its the first in the Type bounds that the type is replaced with upon type erasure, for efficiency, you should therefore put tagging interfaces (that is, interfaces without methods) at the end of the bounds list.

- **Translating Generic Expressions**
  - When you program a call to a generic method, the compiler inserts casts when the return type has been erased.

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

  The erasure of `getFirst` has return type `Object`. The compiler automatically inserts the cast to `Employee`. That is, the compiler translates the method call into two virtual machine instructions:

  - A call to the raw method `Pair.getFirst`
  - A cast of the returned `Object` to the type `Employee` Casts are also inserted when you access a generic field in the resulting byte code.

- **Translating Generic Methods**
  - Read this section from **Page 427-428 Core Java Vol I**
  - Basically lets say we have a class extending a generic type as below:

```
class DateInterval extends Pair<LocalDate>{
      @Override
      public void setSecond(LocalDate second){
            .. .. ..
      }

}
```

  Now in the above example, Pair class provides a method that is:

```
class Pair<T>{
      public void setSecond(T second){
            .. .. ..
      }
}
```

So the above method is being overridden in the subclass `DateInterval`, but after type erasure, the above method becomes:

```
class Pair{
      public void setSecond(Object second){
            .. .. ..
      }
}
```

Which means that the `DateInterval` class has two different methods with same name and not one method that overrides the method of superclass. To fix this, upon type erasure, the compiler inserts a bridge method where the method with `Object` parameter calls the method that was supposed to override it so as to make it behave in accordance with polymorphism. So the class `DateInterval` after type erasure becomes:

```
class DateInterval extends Pair {//Type Erasure
      //bridge method
      public void setSecond(Object second){
            setSecond((LocalDate) second);
      }

      public void setSecond( LocalDate second){
            .. .. ..
      }
}
```

- Same happens in case of a method overriding a method with a type parameter return type. in the virtual machine, the parameter types *and the return type* specify a method. Therefore, the compiler can produce bytecodes for two methods that differ only in their return type, and the virtual machine will handle this situation correctly.
- In summary, you need to remember these facts about translation of Java generics:
  - There are no generics in the virtual machine, only ordinary classes and methods.
  - All type parameters are replaced by their bounds.
  - Bridge methods are synthesised to preserve polymorphism.
  - Casts are inserted as necessary to preserve type safety.

- **Calling Legacy Code**
  - When Java generics were designed, a major goal was to allow interoperability between generics and legacy code.
  - Basically when you pass a generic object to a legacy code, the compiler issues a warning because there is no assurance what the code might do with the object because to be able to allow generics to operate with legacy code, the type assurance that the generic provided is erased in the byte codes and you might get back or the method might mutate

the object with values of type other than the one specified for that object. In such a case, future operations on the object may cause bad cast exceptions.

- There's nothing much you can do here except:
  - If the purpose of passing an object to the method is just to give it read-only access to the values in the object and not get mutated results in the object, pass a clone of the object.
  - If indeed mutated results are expected then make sure that the resulting values conform to the specified type for the generic.
  - When using generics with legacy code, theres not much you can do about the warnings, so take care of the steps above and add the following annotation over such a variable or method using such a variable:

```
@SuppressWarnings("unchecked")
```

- Read this section from **Page 429-430 Core Java Vol I**

- Restrictions and Limitations

- Most of these restrictions are a consequence of type erasure.

- **Type Parameters Cannot be Instantiated with Primitive Types**
  - **You cannot substitute a primitive type for a type parameter.** The reason is of course type erasure. After type erasure the class fields are either of type `Object` or of the bounding type if one is given. And that type can't be used to store primitive types.

- **Runtime Type Inquiries Only work with Raw Types**
  - **Objects in the virtual machine always have a specific non-generic type. Therefore, all type inquiries yield only the raw type**. For example:

```
if (a instanceof Pair<String>)   //Error
if (a instanceof Pair<T>) // Error
```

  The above tests can only test if a is an instance of `Pair` of any type but not of a particular type. Similarly, the following cast generates a warning that it can only check for object a to be of type `Pair` and not `Pair` of `String` specifically:

```
Pair<String> p = (Pair<String>) a;
// Warning--can only test that a is a Pair
```

- To remind you of the risk, you will get a compiler error (with `instanceof`) or warning (with casts) when you try to inquire whether an object belongs to a generic type. In the same spirit, the `getClass` method always returns the raw type:

  ```
  Pair<String> stringPair = . . .;
  Pair<Employee> employeePair = . . .;
  // the return values below are equal
  if (stringPair.getClass() == employeePair.getClass())
  ```

  The comparison yields true because both calls to `getClass` return `Pair.class`.

- **You Cannot Create Arrays of Parametrised Types**
  - **Arrays of parametrised types are outlawed**. You can declare a variable of type `Pair<String>[]`. But you can't initialise it with a `new Pair<String>[10]`.

    ```
    Pair<String>[] table = new Pair<String>[10]; // Error
    ```

  - After erasure, the type of `table` is `Pair[]`. You can convert it to `Object[]`:

    ```
    Object[] objectArray = table;
    ```

    An array remembers its component type and throws an `ArrayStoreException` if you try to store an element of the wrong type:

    ```
    objectArray[0] = "Hello"; // Error--component type is Pair
    ```

    But erasure renders this mechanism ineffective for generic types. The assignment :

    ```
    objectArray[0] = new Pair<Employee>();
    ```

    would pass the array store check but still result in a type error. For this reason, arrays of parameterised types are outlawed. only the creation of these arrays is outlawed. You can declare a variable of type `Pair<String>[]`. But you can't initialise it with a new `Pair<String>[10]`.

  - You can declare arrays of wildcard types and then cast them:

    ```
    Pair<String>[] table = (Pair<String>[]) new Pair<?>[10];
    ```

    The result is not safe. If you store a `Pair<Employee>` in `table[0]` and then call a String method on `table[0].getFirst()`, you get a `ClassCastException`.

  - If you need to collect parameterised type objects, simply use an `ArrayList`: `ArrayList<Pair<String>>` is safe and effective.
  - For more, Read **Page 431-432 Core Java Vol 1**.

- **Var-args Warnings**

- Java doesn't support arrays of generic types. In this section, we discuss a related issue: passing instances of a generic type to a method with a variable number of arguments. Consider the method below:

```
public static <T> void addAll(Collection<T> coll, T... ts) {
     //parameter ts is actually an array that holds all
     //supplied arguments
     for (t : ts){
          coll.add(t);
     }
}
```

  Now consider this call:

```
Collection<Pair<String>> table = . . .;
Pair<String> pair1 = . . .;
Pair<String> pair2 = . . .;
addAll(table, pair1, pair2);
```

  In order to call this method, the Java virtual machine must make an array of `Pair<String>`, which is against the rules. However, the rules have been relaxed for this situation, and you only get a warning, not an error.

- You can suppress the warning in one of two ways. You can add the annotation `@SuppressWarnings("unchecked")` to the method containing the call to `addAll`. Or, as of Java SE 7, you can annotate the `addAll` method itself with `@SafeVarargs`:

```
 @SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

  This method can now be called with generic types. You can use this annotation for any methods that merely read the elements of the parameter array.

- You can use the `@SafeVarargs` annotation to defeat the restriction against generic array creation, using this method:

```
@SafeVarargs
static <E> E[] array(E... array) {
     return array; }
```

  Now you can call :

```
Pair<String>[] table = array(pair1, pair2);
```

  This seems convenient, but there is a hidden danger. The code :

```
Object[] objectArray = table;
objectArray[0] = new Pair<Employee>();
```

  will run without an `ArrayStoreException` (because the array store only checks the erased type), and you'll get an exception elsewhere when you work with `table[0]`.

Tuesday, 7 March 2017

- **Read the following sections from the book : Page 430-440 Core Java Vol 1**
  - **You Cannot Instantiate Type Variables**
    - **You cannot use type variables in an expression such as** `new T(…)`. Type erasure would change `T` to `Object`, and surely you don't want to call `new Object()`.
    - The best workaround, available since Java SE 8, is to *make the caller provide a constructor expression.* For example:

```
Pair<String> p = Pair.makePair(String::new);
```

The makePair method receives a `Supplier<T>`, the functional interface for a function with no arguments and a result of type `T`:

```
public static <T> Pair<T> makePair(Supplier<T> constr) {
     return new Pair<>(constr.get(), constr.get());
}
```

A more traditional workaround is to construct generic objects through reflection, by calling the `Class.newInstance` method. Still you cannot call:

```
T.class.newInstance(); //Error
```

as that would be erased to `Object.class.` Instead, you must design the API so that you are handed a `Class` object, like this:

```
public static <T> Pair<T> makePair(Class<T> cl) {

     try {
          return new Pair<>(
                      cl.newInstance(), cl.newInstance());
     }
     catch (Exception ex) {
          return null;
     }
}
```

This method could be called as follows:

```
Pair<String> p = Pair.makePair(String.class);
```

  - **You Cannot Construct a Generic Array**

- You cannot instantiate a generic array. The reason is that an array carries a type, which is used to monitor array stores in the virtual machine. In case of generic arrays, that type would be erased. That is:

```
new T[2]; // not allowed
```

- Read more about it in detail in the book **Page 435-436 Core Java Vol 1**. **[IMPORTANT]**

- **Type Variables Are Not Valid in Static Contexts of Generic Classes**
  - **You cannot reference type variables in static fields or methods.**

```
public class Singleton<T> {
   private static T singleInstance; // Error
   public static T getSingleInstance() // Error {

   if (singleInstance == null) {
        // construct new instance of T
   }

   return singleInstance; }
}
```

  - If the above could be done, then a program could declare a `Singleton<Random>` to share a random number generator and a `Singleton<JFileChooser>` to share a file chooser dialog. But it can't work.
  - After type erasure there is only one `Singleton` class, and only one `singleInstance` field. For that reason, **static fields and methods with type variables are simply outlawed.**

- **You Cannot Throw or Catch Instances of a Generic Class**
  - **You can neither throw nor catch objects of a generic class**. In fact, it is not even legal for a generic class to extend `Throwable`.

```
// Error--can't extend Throwable
public class Problem<T> extends Exception {
     //Some code
}
```

  - **You CANNOT use a type variable in a catch clause. That is you cannot catch type parameter.**

```
public static <T extends Throwable> void doWork(Class<T> t){
     try {
          //do work
     }
     catch (T e) // Error--can't catch type variable {
```

```
                Logger.global.info(...)
            }
        }
```

- **You CAN Although throw a Type parameter**. And so, it is OK to use type variables in exception specifications.

```
public static <T extends Throwable> void
                              doWork(T t) throws T // OK {

      try {
            do work
      }
      catch (Throwable realCause) {
            t.initCause(realCause);
            throw t;   //OK
      }
}
```

- **You Can Defeat Checked Exception Checking**
  - A bedrock principle of Java exception handling is that you must provide a handler for all checked exceptions. You can use generics to defeat this scheme.

```
@SuppressWarnings("unchecked")
public static <T extends Throwable> void
                              throwAs(Throwable e) throws T {
      throw (T) e;
}
```

  Suppose this method is contained in a class `Block`. When you call:

```
Block.<RuntimeException>throwAs(t);
```

  then the compiler will believe that `t` becomes an unchecked exception. The following turns all exceptions into those that the compiler believes to be unchecked:

```
try {
      //do work
}
catch (Throwable t) {
      Block.<RuntimeException>throwAs(t);
}
```

  - Normally, you have to catch all checked exceptions that can be thrown inside a method and wrap them into unchecked exceptions—the method is then declared to throw no checked exceptions. But here, we don't wrap. We simply throw the exception, tricking the compiler into believing that it is not a checked exception. Using generic classes, erasure, and the `@SuppressWarnings` annotation, we were able to defeat an essential part of the Java type system.

- **Beware of Clashes after Erasure**
  - Sometimes it may so happen that upon erasure, a generic method may class with an already existing method. In such a case, we have to rename the offending method. Example:

    ```
    public class Pair<T> {
       public boolean equals(T value) {
            return first.equals(value)
                         && second.equals(value);
       }
       ...
    }
    ```

    Upon type erasure, the above method becomes:

    ```
       public boolean equals(Object value)
       //clashes with Object.equals
    ```
  - **To support translation by erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterisations of the same interface.**
  - For example, the **following is illegal**:

    ```
    class Employee implements Comparable<Employee> { . . . }
    class Manager extends Employee implements
                              Comparable<Manager> {. . .} //Error
    ```

    `Manager` would then implement both `Comparable<Employee>` and `Comparable<Manager>`, which are different parameterisations of the same interface.
  - All this while, the non-generic version is still legal. The reason is that for generics: T**here would be a conflict with the synthesised bridge methods**. A class that implements `Comparable<X>` gets a bridge method

    ```
       public int compareTo(Object other) {
            return compareTo((X) other);
       }
    ```

    You cannot have two such methods for different types X.

- **Inheritance Rules for Generic Types**

- In general, there is *no* relationship between `GenericClass<S>` and `GenericClass<T>`, no matter how `S` and `T` are related. That is, consider a class and a subclass, such as `Employee` and `Manager`. No matter their relationship: `Pair<Manager>` is not be a subclass of `Pair<Employee>`. Hence, the following code will not compile:

```
Manager[] topHonchos = . . .;
Pair<Employee> result = ArrayAlg.minmax(topHonchos);//Error
```

  The `minmax` method above returns a `Pair<Manager>`, not a `Pair<Employee>`, and it is illegal to assign one to the other.

- The rule is necessary for type safety. Suppose we were allowed to convert a `Pair<Manager>` to a `Pair<Employee>`. Consider this code:

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies;
//illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

  Clearly, the last statement is legal. But `employeeBuddies` and `managerBuddies` refer to the *same object*. We now managed to pair up the CFO with a lowly employee, which should not be possible for a `Pair<Manager>`.

- You just saw an important difference between generic types and Java arrays. You can assign a `Manager[]` array to a variable of type `Employee[]`:

```
Manager[] managerBuddies = { CEO, CFO };
Employee[] employeeBuddies = managerBuddies; // OK
```

  However, arrays come with special protection. If you try to store a lowly employee into `employeeBuddies[0]`, the virtual machine throws an `ArrayStoreException`.

- You can always convert a parameterised type to a raw type. For example, `Pair<Employee>` is a subtype of the raw type `Pair`. This conversion is necessary for interfacing with legacy code

- Unfortunately, you can also cause a type error by doing so. Consider this example:

```
Pair<Manager> managerBuddies = new Pair<>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(". . .")); // only a compile-time
                                        // warning
```

- However, keep in mind that you are no worse off than you were with older versions of Java. The security of the virtual machine is not at stake. When the foreign object is retrieved with `getFirst` and assigned to a `Manager` variable, a `ClassCastException` is thrown, just as in the good old days. You merely lose the added safety that generic programming normally provides.

- Finally, generic classes can extend or implement other generic classes. In this regard, they are no different from ordinary classes. For example, the class `ArrayList<T>` implements the interface `List<T>`. That means an `ArrayList<Manager>` can be converted to a `List<Manager>`. However, as you just saw, an `ArrayList<Manager>` is *not* an `ArrayList<Employee>` or `List<Employee>`.

- **Wildcard Types**
  - **Read from Page 442-454 Core Java Vol 1. You need context to understand the concepts.**

- **Collections**
  - **Read from Book Core Java Vol 1.**

- **Deploying Java Applications**
- **JAR Files**
  - When you package your application, you want to give your users a single file, not a directory structure filled with class files. Java Archive (JAR) files were designed for this purpose. A JAR file can contain both class files and other file types such as image and sound files. Moreover, JAR files are compressed, using the familiar ZIP compression format.
  - **Creating JAR Files**
    - Use the jar tool to make JAR files. (In the default JDK installation, it's in the *jdk*/bin directory.) The most common command to make a new JAR file uses the following syntax:
    -