**III. Java Basic**

1. **Platform**

* **J2SE**: standard Edition
* **J2EE**: for Enterprise Application
* **J2ME**: for Mobile Application

1. **Feature**

* **Objected Oriented VS Procedure Oriented**
* Easier manage when code size up
* Provide data hiding by using access modifier
* Simulate real-world event more efficiently
* **Platform independent**: compile into platform independent byte code by JVM
* **Polymorphism**: one task is performed different ways (override, overload)
* **Encapsulation**: wrap code and data together into a single unit
* **Inheritance:** one object acquire all properties and behaviors of parent object
* **Abstraction**: hide internal detail and show functionality

1. **Primitive Data Types**

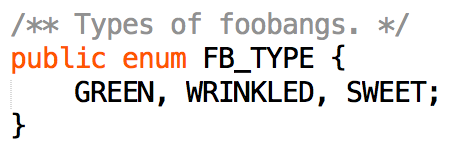
* **Byte**: 8-bit signed two’s complement integer (-2^7 --- 2^7-1)
* **Short**: 16-bit
* **Int**: 32-bit
* **Long**: 64-bit
* **Float**: single-precision 32-bit (default: 0.0f)
* **Double**: double-precision 64-bit (default: 0.0d)
* **Boolean**
* **Char**: 16-bit Unicode character

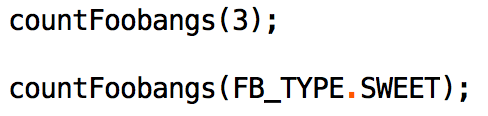
1. **Numbers Class**

* **Wrapper** **classes**: when need to use objects instead of primitive data types
* **Boxing**: convert primitive data types into object
* **Int to String**:
* String.valueOf(int): Returns a String object holding the value of int.
* Integer.toString(int)
* **String to int**:
* Integer.parseInt(String): Parses the string as int.
* Integer.valueOf(String): Returns an Integer object.
* **XxxxValue():** return the value of this Number object to the primitive data type
* **Xxxxxxx.valueOf(char):** return object holding
* **Character**
* Character.isLetter(char)
* Character.isDigit(char)
* Character.isWhiteSpace(char)
* Character.isUpperCase(char)
* Character.toLowerCase(char)
* Character.toString(char)
* **BigDecimal**
* Immutable, arbitrary-precision signed decimal numbers.
* Add(BigDecimal)
* Substract(BigDecimal)
* Multiply(BigDecimal)
* divide(BigDecimal)
* compareTo(BigDecimal): scale matters!
* int scale()
* doubleValue()
* **String**
* charAt(int)
* compareTo(String)
* Boolean StartsWith(String)/ EndsWith()
* Equals(String)
* IndexOf(char/String)
* Matches(String)
* ReplaceAll(regex, repalcement)
* String[] split
* substring(int, int)
* Char[] toCharArray()
* toLowerCase()
* trim()

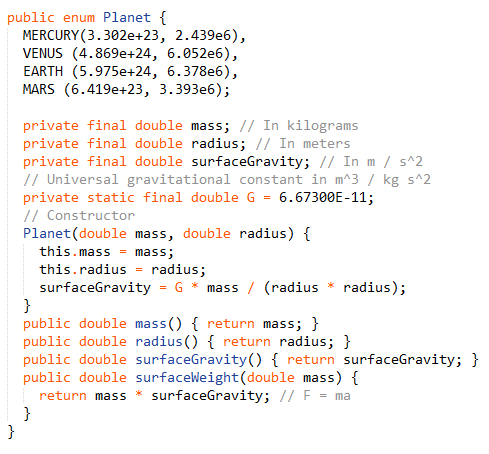
1. **Enum Type**

* A type whose legal values consist of a fixed set of constants
* Always use enums when a variable can only take one out of a small set of possible values
* Add compile-time check and avoid errors from passing in invalid constants
* Use EnumSet instead of bit fields
* **Examples**





or



1. **Arrays Class**

* Arrays.asList(xxx[])
* Arrays.binarySearch(sorted array, key)
* Arrays.copyOf(xxx[], newLength)
* Arrays.equals(xxx[], xxx[])
* Arrays.fill(xxx[], value)
* Arrays.sort(xxx[])
* Arrays.toString(xxx[])

1. **Variable:**

* **Local variables**: declared in methods, constructors, or blocks
* **Instance variables**: declared in a class, but outside a method, constructor or any block
* **Static variables**

1. **Static keyword**

* static members belong to the class instead of a specific instance
* Cannot override static method
* Static methods belong to the class and not the objects, so the concept of overriding for runtime polymorphism is not applicable.
* **static class**: only if it is a nested class, and can be accessed without having an object of outer class.
* **static method**:
* belong to the class (not an instance of it).
* Used in utility classes, and use no instance variables
* usually take input from the parameters, perform actions on it, then return result
* **static variables**
* have a single value for all instance of a class
* static variables are often used for constants, avoid hard-code

1. **Access Control Modifiers:**

* **Private**: visible to the class only
* **Public**: visible to the world
* **Protected**: visible to the package and all subclass
* **No modifier**: visible to the package

1. **Files and I/O**

* **Stream**: a sequence of data
* **Standard Stream**: System.in/ System.out/ System.err
* **Byte Stream**
* FileInputStream/FileOutputStream
* perform input/output of bytes
* **Character Stream**
* FileReader/ FileWriter
* a general tool to read in characters from a File
* perform input/output of 16-bit Unicode
* **Buffered Stream**
  + BufferredReader/ BufferedWriter
  + unbuffered I/O can lead to overhead with disk access
  + buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.
  + similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.
  + The BufferedReader class can wrap around Readers, to buffer the input and improve efficiency.
  + *BufferedReader br = new BufferedReader(new FileReader("ride.in"));*

*Scanner sc = new Scanner(br);*

*PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("ride.out")));*

*pw.println("GO");*

1. **StringTokenizer, String.split(), Scanner**

* **StringTokenizer** 
  + StringTokenizer st = new StringTokenizer ("this is a test");

while (st.hasMoreTokens()) {

System.out.println(st.nextToken());

}

* **String.split()**
  + String[] split(String regex)
* **Scanner**
  + Scanner sc = new Scanner(

new File("myNumbers")).useDelimiter("\\s\*fish\\s\*");

while (sc.hasNextLong()) {

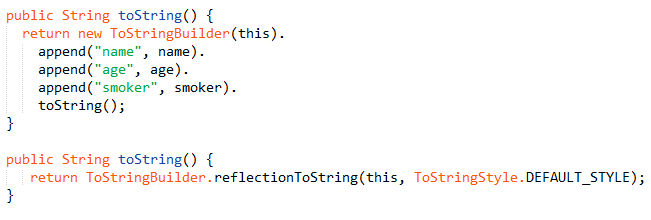
long aLong = sc.nextLong();

}

* **Comparison between split(), Scanner, StringTokenizer**
  + Scanner: designed for cases where you need to parse a string, pulling out data of different types. It's very flexible.
  + String.split(): Splits a string into string array with given regex expression.
  + StringTokenizer: designed for pulling out tokens delimited by fixed substrings. Because of this restriction, it's about twice as fast as String.split().

1. **ToStringBuilder Class**

* This class enables a good and consistent toString() to be built for any class or object.



1. **Optional Class**

* A container object which helps to indicate the absence of a return value, when the method could return null
* Check null after method call



* Check null before return



1. **Exception:**

* **Definition**: an event occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
* **throw exception**: Creating an exception object and handing it to the system
* throw by method: public void writeList() throws IOException{ … }
* throw statement: throw new IndexOutOfBoundsException ();
* **try block**: where exception can occur
* **catch block**: exception handler that handle a particular type of exception.
* **finally block**: code that is guaranteed to execute even if an exception occurs, and is the right place to close files, recover resources
* **Catch or Throw?**
* You generally catch an exception in a method when you want your program to continue running.
* You throw an exception when you want a higher level method that is calling that method to handle the exception instead.
* **Advantages of Exception**
* Separating Error-Handling Code from "Regular" Code
* Propagating Errors Up the Call Stack
* Grouping and Differentiating Error Types

Xxx, (a,b) -> (a.comparet)

1. **Lambda Expression**

* **Substitute of anonymous inner class**
* An inner class declared without a class name
* a way to implement classes that occur only once
* Lambdas transfer anonymous inner classes into 1-line code
* **Syntax**
* (argument-list) -> {body}
* Must use return keyword when lambda contains multiple statements
* **Ideal use case**
* Use as Predicate
* Predicate is a functional interface and therefore can be used as assignment target for lambda expression or method reference
* Predicate<Person> adult = p -> p.getAge() > 18
* Looping
* personList.forEach(Person::printLastName);
* Chaining and Filters
* List<Client> result = clients.stream().filter(c -> (users.stream().map(User::getName).collect(Collectors.toList())
* **Method references**
* When lambda expression does nothing but call an existing method, it's clearer to refer to the existing method by name.
* Reference to a static method

ContainingClass::staticMethodName

* Reference to an instance method of a object

containingObject::instanceMethodName

* Reference to an instance method of an arbitrary object of a particular type

ContainingType::methodName

* Reference to a constructor

ClassName::new

1. **Stream Class**

* **Support sequential and parallel aggregate operations**
* To perform a computation, stream [operations](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps) are composed into a *stream pipeline*.
* A stream pipeline consists of a source
* zero or more *intermediate operations*
* a *terminal operation*
* produces a result or side-effectas [count()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#count--) or [forEach(Consumer)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach-java.util.function.Consumer-)
* **Streams are lazy**
* computation on the source data is only performed when the terminal operation is initiated
* **Methods**
* Boolean noneMatch()/allMatch()/anyMatch(Predicate<T>)
* Predicate example: Predicate<T> p = e -> e.id < 10000
* Static <T> Stream.Builder<T> Builder()
* <R,A> R Collect(Collector<T,A,R>)
* List<String> asList = stringStream.collect(Collectors.toList());
* Map<String, List<Person>> peopleByCity = personStream.collect(Collectors.groupingBy(Person::getCity));
* Long count()
* Optional<T> findAny()/findFisrst()
* Optional<T> max(Comparator)/min()
* Stream<R> map(Function)
* Stream<R> flatMap(Function<T, Stream>)
* T reduce(T, BinaryOperator)
* Void forEach(Consumer<T> action)
* Stream<T> distinct()
* Stream<T> filter(Predicate<T>)
* Stream<T> limit(long)
* Stream<T> of(T … values)
* Stream<T> peek(Consumer<T> action)
* Stream<T> skip(long)
* Stream<T> sorted()
* Object[] toArray()

1. **Super keyword:**

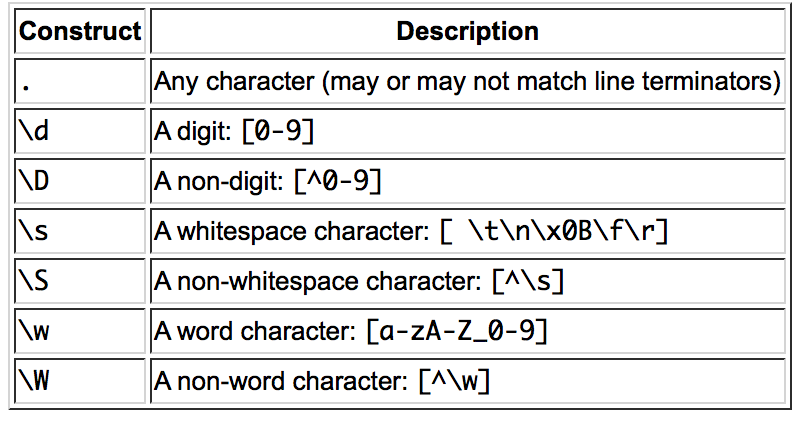
* differentiate the members of superclass from the members of subclass, if they have same names
* super.variable
* super.method();
* invoke the superclass constructor from subclass.
* super(values);

1. **instanceof keyword**

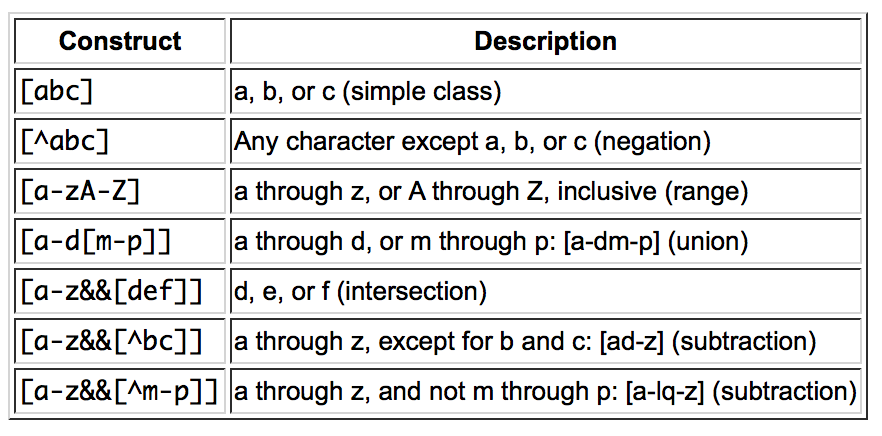
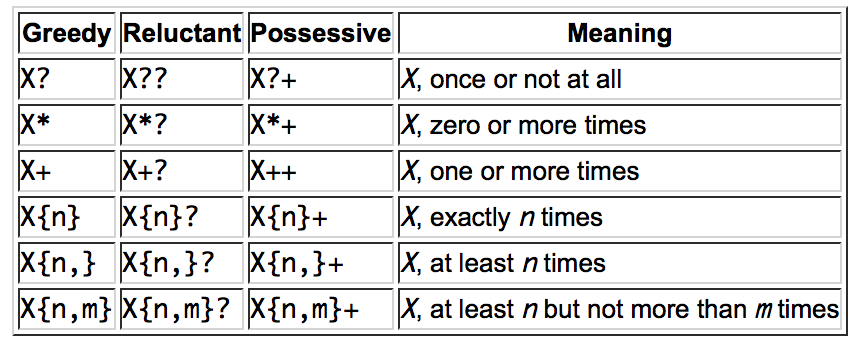
* test if an object is of a specified type.
* if (objectReference instanceof type)

1. **Regular expressions**

* a way to describe a set of strings based on common characteristics
* can be used to search, edit, or manipulate text and data
* predefined character classes: shorthand for commonly used regular expressions



* **metacharacter**: a special character that affects the way a pattern is matched
* Precede the metacharacter with a backslash (\) 🡪 ordinary character
* **quantifiers**



1. **Interface & Abstract class**

* **abstract class**: a class can only be instantiated by creating an inheriting subclass
* **Interface**: similar to abstract class, it is a collection of abstract methods.
* abstract class ensure subclass to implement abstract class to provide the minimum functionality to work
* **Differences**
* Interface is more about behavior
* Abstract class can have both abstract and concrete methods (must declare methods with ‘abstract’); interface only abstract
* A class that implements an interface must implement all the methods declared in the interface; while usually same for abstract class, except subclass is also be declared as abstract
* Interface can only have static final variable
* How to choose
* **Consider using abstract classes if:**
* You want to share code among several closely related classes.
* You want to declare non-static or non-final fields.
* **Consider using interfaces if:**
* You expect that unrelated classes would implement your interface.
* You want to take advantage of multiple inheritance of type.

1. **final keyword, finally keyword, finalize method**

* **final**
* variable: cannot be change once initiated
* method: cannot be overridden by subclass
* class: the class cannot be subclassed
* **finally**
* associate with a try/catch block and guarantees that a section of code will be executed even if an exception is thrown
* the block will be executed after try/catch but before return control
* **finalize**
* automatic garbage collector calls finalize () method just before actually destroying the object.
* A class can override finalize() method from Object class to define custom behavior during garbage collection.

1. **Difference between overriding and hiding methods**

* **Overloading**: when 2 methods have same name but differ in type or number of arguments
* If a subclass defines a static method with same signature as the static method in superclass, the the method in subclass hides the one in superclass, o/w override
* The overridden method gets invoked is the one in the subclass.
* But hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass

1. **Generic**

* Generics enable types to be parameters when defining classes, interface and and methods
* **Why use Generics?**
* Stronger type checks at compile time (easier than fixing runtime errors)
* Elimination of casts
* Enable programmers to implement generic algorithm
* Exception
* So don’t use raw types except instanceof operator. Because generic type information is erased at runtime
* **generic type**
* generic type: class or interface that is parameterized over types
* raw type: a generic class or interface without any type arguments
* if assign a raw type to a parameterized type, lead to warning: unchecked conversion
* **generic method**
* Generic methods are methods that introduce their own type parameters
* public <K, V> Boolean compare (Pair<K, V> p1, Pair<K, V> p2)
* **Wildcards**
* Use question mark (?) to represents an unknown type
* upper bounded wildcard
* to relax the restrictions on a variable
* matches a type or any of its subclasses: List<? extends Foo>
* use for "in" variable (source)
* unbounded wildcard
* List<?>
* Use under two scenarios:

1.writing a method that can be implemented using functionality provided in the Object class.

2.When the code is using methods in the generic class that don't depend on the type parameter, etc. size ()

* lower bounded wildcard
* List<? super Integer> list
* cannot specify both upper bound and lower bound wildcard
* use for "out" variable (output)
* **Bounded Type Parameters vs Wildcard**
* Both restrict the types that can be used as arguments
* Difference in bound
* A wildcard can have only one bound, while a type parameter can have several bounds
* A wildcard can have a lower or an upper bound, while there is only upper bound for a type parameter.
* Difference in Syntax
* Type parameter bound

<T extends Class & Interface1 & … & InterfaceN>

* Wildcard bound

upper bound <? extends SuperType>

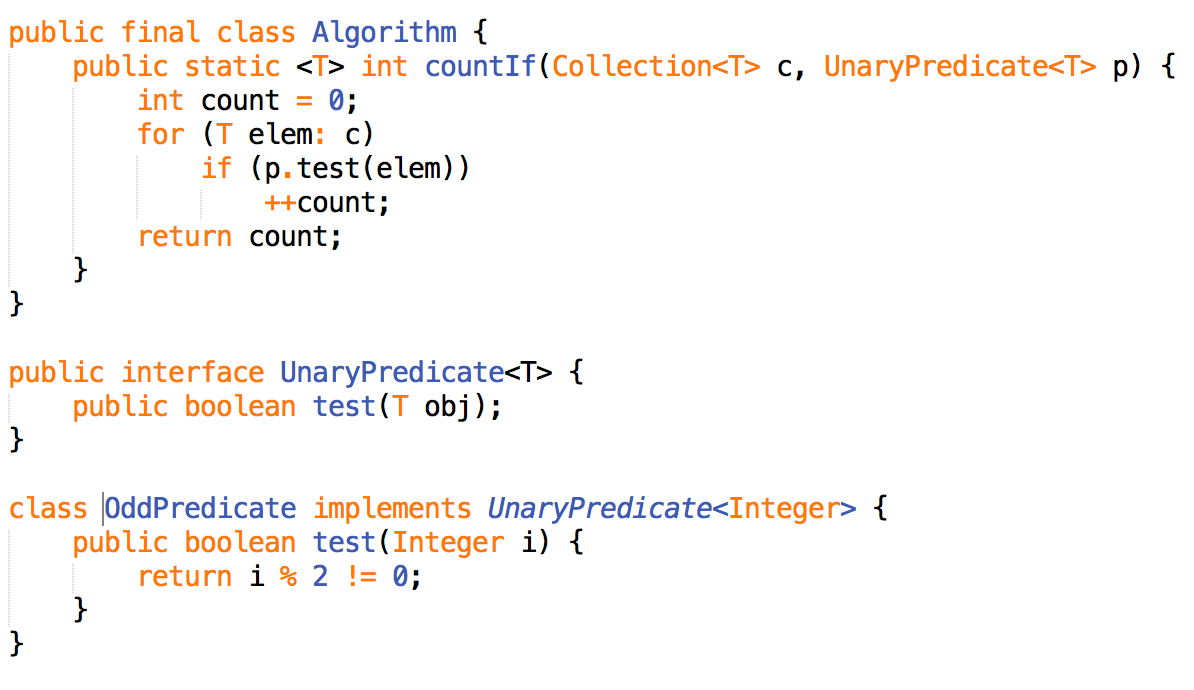
lower bound <? super SubType>

* How to choose
* Use type parameter bound **only** when

public <T extends Shape> void xxMethod(List<T> shapes, T shape) { … }

* o/w, use wildcard

public void drawAll(List<? extends Shape> shapes){ … }

* **Type Erasure**
* if the type parameter is bounded, Java compiler erases all type parameters and replaces each with its first bound
* if the type parameter is unbounded, replaces with Object
* **Eliminate unchecked warnings**
* Iff you can’t eliminate a warning, and you know it is typesafe, suppress the warning with an @SuppressWarnings("unchecked") annotation
* Always use the SuppressWarnings annotation on the smallest scope possible
* **Prefer list to array**
* Arrays are covariant, Generics are not
* Valid and will compile: Object[] arr = new String[10];
* Invalid: List<Object> list = new ArrayList<String>();
* Type unsafe
* generic types don't have any type information at runtime due to type erasure, which lead to type unsafe
* **Restrictions on generics**
* cannot instantiate with primitive types
* cannot create arrays of parameterized types
* Node<Circle> is not a subtype of Node<Shape>
* **Question 1:** Write a generic method to count the number of elements in a collection that have a specific property.

1. **Iterator**

* **Iterable Interface**
* If a collection is iterable, then it can be iterated using an iterator
* **Iterator Interface**
* iterator is the actual object that will iterate through the collection.
* **Advantage over get() method**
* Iterator is much faster than get(index) when list is long
* Use when access list frequently
* **Iterator VS ListIterator**
* Methods of Iterator
* hasNext()
* next()
* remove(): follow after next() and only call once each next()
* Methods of ListIterator
* hasNext()/hasPrevious()
* previous()/next():Can traverse forward
* previousIndex()/nextIndex(): Can obtain index while iterating
* Add(E): Can add element
* Set(E): Can replace existing element value
* Remove()

1. **Math Class**

* performing basic numeric operations
* **Methods**
* Math.abs(int/long/double)
* double Math.exp(double):
* double Math.ceil(double d): return the smallest double >= d
* double Math.floor(double d): return the largest double <= d
* double Math.log(double): base on e
* Math.max(E, E)
* Math.min(E, E)
* double Math.pow(double a, double b): a -> base, b -> exponent
* double Math.rint(double): same as round() except return type
* int Math.round(double/float)
* double Math.sqrt(double)
* double Math.random(): return double >= 0.0, < 1.0
* **Create Random Number**
* int randomNum = rand.nextInt((max - min) + 1) + min
* Math.random() \* (max – min + 1) + min;
* Reservoir sampling (get a random element from unknown-size set)

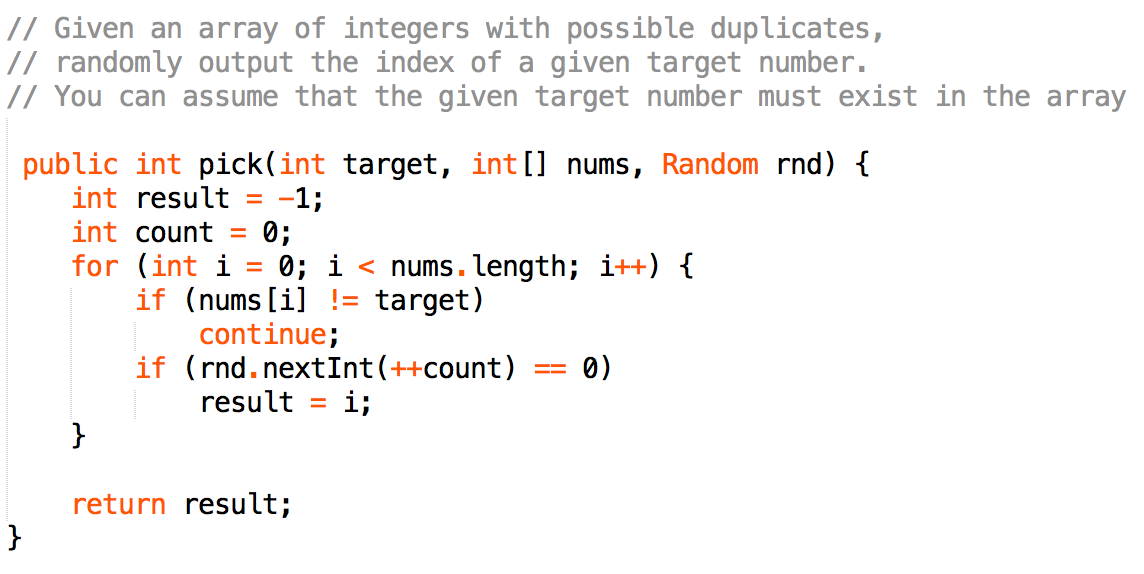
從S中抽取首k項放入「水塘」中

對於每一個S[j]項（j ≥ k）：

隨機產生一個範圍從0到j的整數r

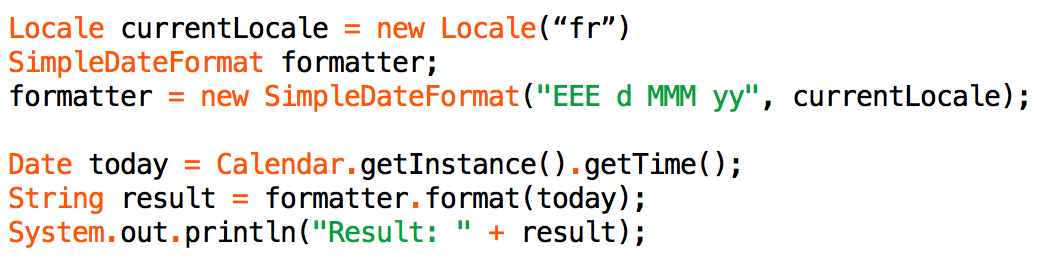
若 r < k 則把水塘中的第r項換成S[j]項

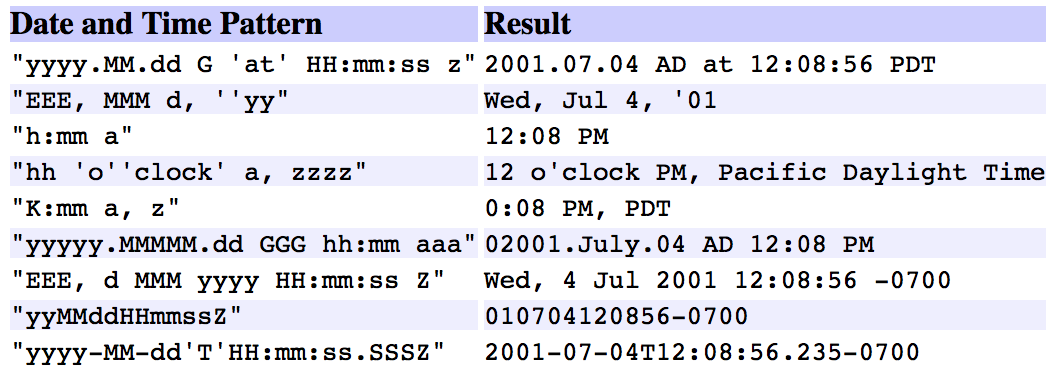
Example



1. **Data & Calendar Class**

* **Jave 8 new Date-Time API feature**
* Thread safe
* Numerous utility method for date operation
* Handle time zone (Local Class, Zoned Class)
* **Calendar Class (Abstract)**
* Initial: Calendar c = Calendar.getInstance()/(Locale l)/(TimeZone zon)
* Set Timezone: c.setTimeZone(tz)
* Set Date: c.set(int field, int value)
* Get Date: c.get(int field)
* Add Date: c.add(int field, int amount)
* Compare Date: after/before(Object when)
* Calendar to Date: Date d = c.getTime()
* Date to Calendar: c.setTime(d)
* Calendar to relative time: long t = c.getTimeInMillis()
* Relative time to Calendar: c.setTimeInMillis(t)
* toString()
* Common field (static final int)
* Calendar.YEAR——年份
* Calendar.MONTH——月份(0-11)
* Calendar.DATE——日期(1-7)
* Calendar.DAY\_OF\_MONTH——和上面的字段完全相同
* Calendar.HOUR——12小时制的小时数
* Calendar.HOUR\_OF\_DAY——24小时制的小时数
* Calendar.MINUTE——分钟
* Calendar.SECOND——秒
* Calendar.DAY\_OF\_WEEK——星期几
* **Date Class**
* Initial: Date()/Date(long date)
* boolean after(Date when)
* compareTo(Date date)
* equals(Object)
* long getTime():Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object
* setTime(long time)
* toString()
* **SimpleDataFormat Class**





1. **Reflection**

* **Definition**:Reflection is a language's ability to inspect and dynamically call classes, methods, attributes.
* **Example**: all objects in Java have the method getClass(), which lets you determine the object's class even if you don't know it at compile time.
* **Why important**: it allow write programs without have to "know" everything at compile time, making them more dynamic