SAMUEL GINN
COLLEGE OF ENGINEERING

Generality

List search

Search space: A list of items that can be compared to each other.

Goal: Locate an item with a specific value or discover that it is not there.

Version A: Given an array of ints, return the index of the first occurrence of a target value in the array or return -1 if the target value is not in the array.

Version B: Given list of songs in a playlist, return the index of the first occurrence of a target song in the list or return -1 if the target song is not in the list.

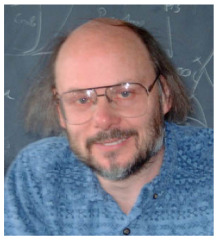
Version C: Given a directory of image files on a disk, return the index of the first occurrence of a target image in the list or return -1 if the target image is not in the directory.

List search

```
public int search(int[] a, int target) {
   int i = 0;
   while ((i < a.length) && (a[i] != target))
      i++;
   if (i < a.length)  return i;
   else               return -1;
}
```

This solves "Version A" of the list search problem, but we should write the code so that it solves most variations of the problem.

```
public <T> int searchC(List<T> a, T target) {
   int i = 0;
   for (T element : a) {
      if (element.equals(target))   return i;
      i++;
   }
   return -1;
}
```



Bjarne Stroustrup

"Lift algorithms and data structures from concrete examples to their most general and abstract form."

Generalized Programming

Alexander Stepanov

Pioneered, with others, the idea of "generic programming."

Helped design the C++ STL, heavily influenced by Ada generics.

Began exploring and implementing the notion of generalized programming in the 1970s.

"It is easy to overlook the importance of what I discovered. [. . .] The significant thing was that making interfaces general – even if I did not quite know what it meant – I made them much more robust. The changes in the surrounding code or changes in the grammar of the input language did not affect the general functions: 95% of the code was impervious to change. In other words: ***decomposing an application into a collection of general purpose algorithms and data structures makes it robust****. (But even without generality, code is much more robust when it is decomposed into small functions with clean interfaces.) Later on, I discovered the following fact: as the size of application grows so does the percentage of the general code. I believe, for example, that in most modern desktop applications the non-general code should be well under 1%."*

Notes on Programming, Alexander Stepanov, October 2007

Making code general

```
public int search(int[] a, int target) {
   int i = 0;
   while ((i < a.length) && (a[i] != target))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Two basic issues to address in making this code more general:

The data type being used

How the "list" is represented

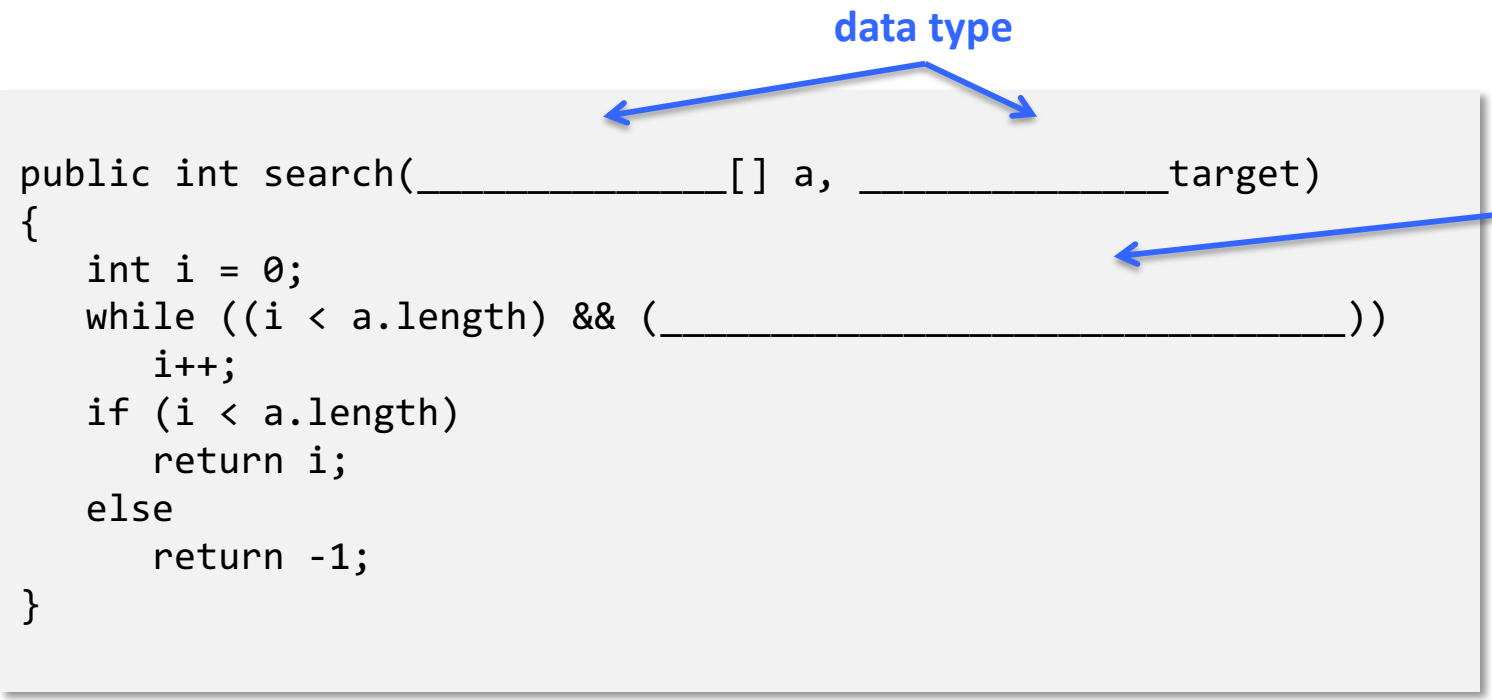
What we'll do: Choose a very general data type but retain type-safety.

Select a representation for the "list" that's less concrete and more flexible.

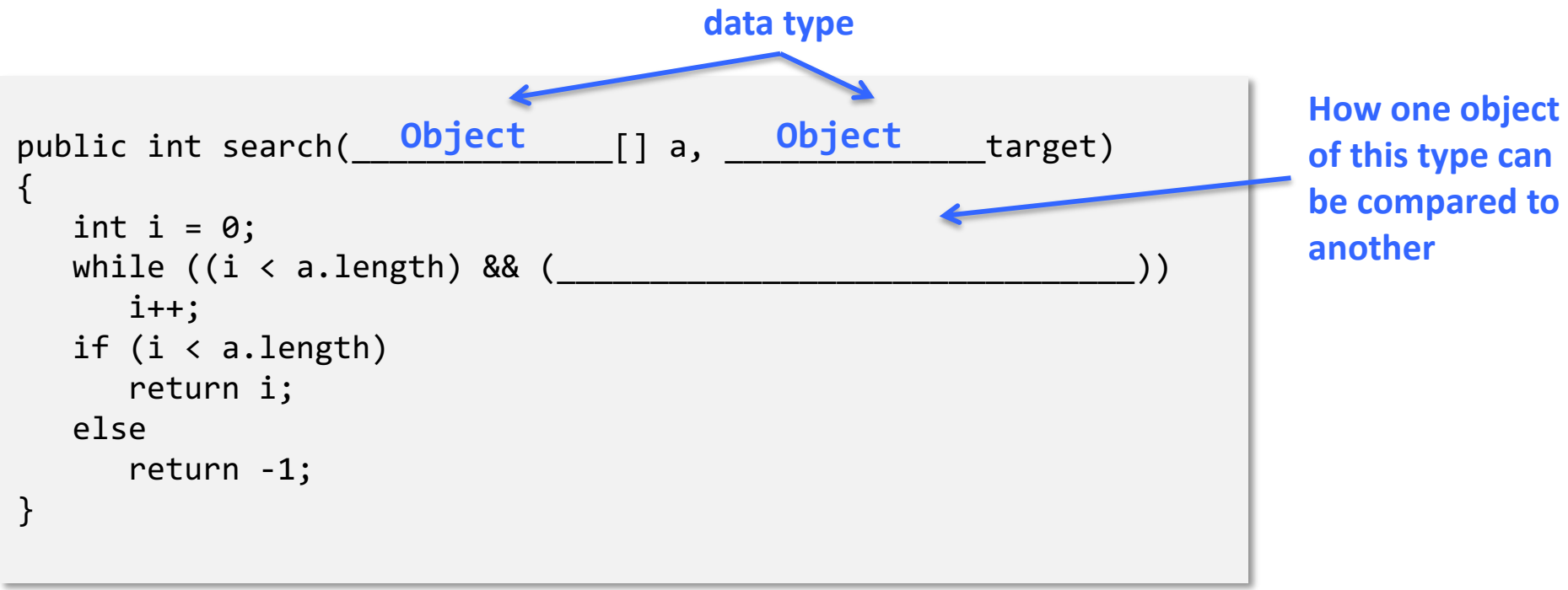
Employ general programming idioms that tend to make code more reusable.

Generalizing type – Object

General data types



General data types



General data types

data type

```
public int search(__Object_____[] a, __Object_____target)
{
   int i = 0;
   while ((i < a.length) && (_____________________________))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

How one object of this type can be compared to another

Q: Which of the choices below would be best to perform this comparison?

A. `a[i] != target`

B. `!a[i].equals(target)` ⬅

C. `a[i].compareTo(target) != 0`

D. `a[i].compareTo(target) < 0`

General data types

data type

```
public int search(__Object_____[] a, __Object_____target)
{
   int i = 0;
   while ((i < a.length) && (__!a[i].equals(target)_____))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

How one object of this type can be compared to another

This code assumes that clients will provide an array of objects that provide an appropriate equals method. It isn't this code's responsibility to decide what equality means.

Client side

```
public void testSearchString() {
    String[] a = {"2", "4", "6", "8", "10"};
    String target = "8";
    int expected = 3;
    int actual = search(a, target);

    assert actual == expected;
 }
```

This will work – String overrides equals.

Client side

```
public static void testSearchInteger() {
     Integer[] a = {2, 4, 6, 8, 10};
     Integer target = 8;
     int expected = 3;
     int actual = search(a, target);

     assert actual == expected;
  }
```

This will work – Integer overrides equals.

Client side

```
public static void testSearchBook() {
     Book[] a = {new Book("author1", "title1", 123),
           new Book("author2", "title2", 456),
           new Book("author3", "title3", 789),
           new Book("author4", "title4", 123),
           new Book("author5", "title5", 456)};
     Book target = new Book("author4", "title4", 999);
     int expected = 3;
     int actual = search(a, target);

     assert actual == expected;
  }
```

This will NOT work as intended UNLESS Book overrides equals to be based on same author, same title.

General data types

data type

```
public int search(____Object____[] a, ____Object____target)
{
   int i = 0;
   while ((i < a.length) && (_____!a[i].equals(target)_____))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

How one object of this type can be compared to another

It's the responsibility of the underlying data class to provide an appropriate implementation of the equals method.

The equals method

The `equals()` method is defined in class Object, so all classes in Java have an `equals()` method.

The default implementation is the same as the == operator (aliasing/object identity).

Java classes such as Integer, Double, String, Date, etc. have customized, overridden implementations of `equals()`.

You will want to override `equals()` when the class you're writing has a notion of **logical equality** that's different from mere object identity, and a superclass hasn't already overridden `equals()` to implement the desired behavior.

```
public class Book {
   private String author;
   private String title;
   private int pages;

   ...
}
```

What does it mean for one book to be equal to another?

```
public boolean equals(Book that) {
   return this.title.equals(that.title);
}
```

Note: This doesn't even override equals()! It just overloads it.

The equals contract

http://download.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one. The equals method implements an **equivalence relation** on non-null object references:

- It is **reflexive**: for any non-null reference value x, x.equals(x) should return true.
- It is **symmetric**: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is **transitive**: for any non-null reference values 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**: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

An equals recipe

```
public class Book {
   private String author;
   private String title;
   private int pages;
   ...

public boolean equals(Object obj) {

   if (obj == this)   return true;

   if (obj == null)   return false;

   if (!(obj instanceof Book))   return false;

   Book that = (Book) obj;

   return this.title.equals(that.title) &&
          this.author.equals(that.author);
}
```

- Must be of type Object to override
- Check for aliasing
- Check for null
- Check for type compatibility
- Cast from Object
- Check that all significant fields are equal

Type compatibility

`instanceof` operator

Compares an object to a specified class or interface.

```
if (!(obj instanceof Book))
                    return false;
```

Allows mixed-type equality within class/interface hierarchies.

Can violate symmetric and transitive properties.

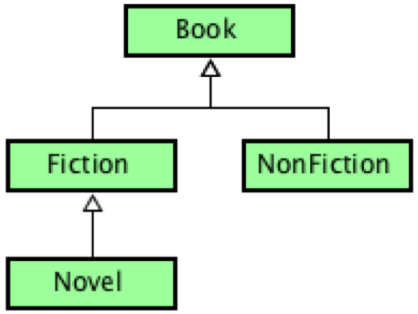
```
Book b = new Book(
             "Nineteen Eighty-Four",
             "George Orwell",
             326);

Fiction f = new Fiction(
             "Nineteen Eighty-Four",
             "George Orwell",
             326,
             "Winston Smith");
```

?

```
b.equals(f)

f.equals(b)
```



Type compatibility

`getClass` method

Defined in Object. Returns the *runtime* class of an object.

```
if (obj.getClass() != this.getClass())
                    return false;
```

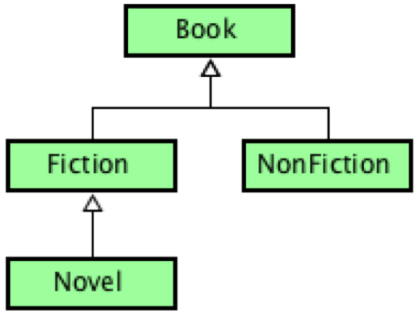
Does not allow mixed-type equality.

Can meet equals contract (will be reflexive, symmetric, and transitive), but could be too inflexible.

```
Book b = new Book(
             "Nineteen Eighty-Four",
             "George Orwell",
             326);

Book f = new Fiction(
             "Nineteen Eighty-Four",
             "George Orwell",
             326,
             "Winston Smith");
```

Neither b nor f would equal the other since they have different runtime types.



Type compatibility

Which should you use: instanceof or getClass()?

There's no single answer to the question for all situations, and it speaks to larger issues involved with designing and using inheritance hierarchies.

Some good advice:

Joshua Bloch

Chief Java Architect at Google

Effective Java, Second Edition, 2008

General data types

```
public int search(Object[] a, Object target) {
   int i = 0;
   while ((i < a.length) && (!a[i].equals(target)))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Pro:
Will work for any reference types.
Always type-safe if the client has implemented equals correctly.

Con:
Must restrict operations to those common to all Objects. (e.g., equals)

If the algorithm being implemented requires any operation that an Object can't perform, this approach to generality doesn't work.

Example: The min method

```
public Object min(Object[] a) { . . . }
```

We need more than equality comparisons to write the min method.

Generalizing type – Comparable

The Comparable interface

http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
public interface Comparable<T>
```

This interface imposes a **total ordering** on the objects of each class that implements it. This ordering is referred to as the class's ***natural ordering***, and the class's compareTo method is referred to as its natural comparison method.

. . .

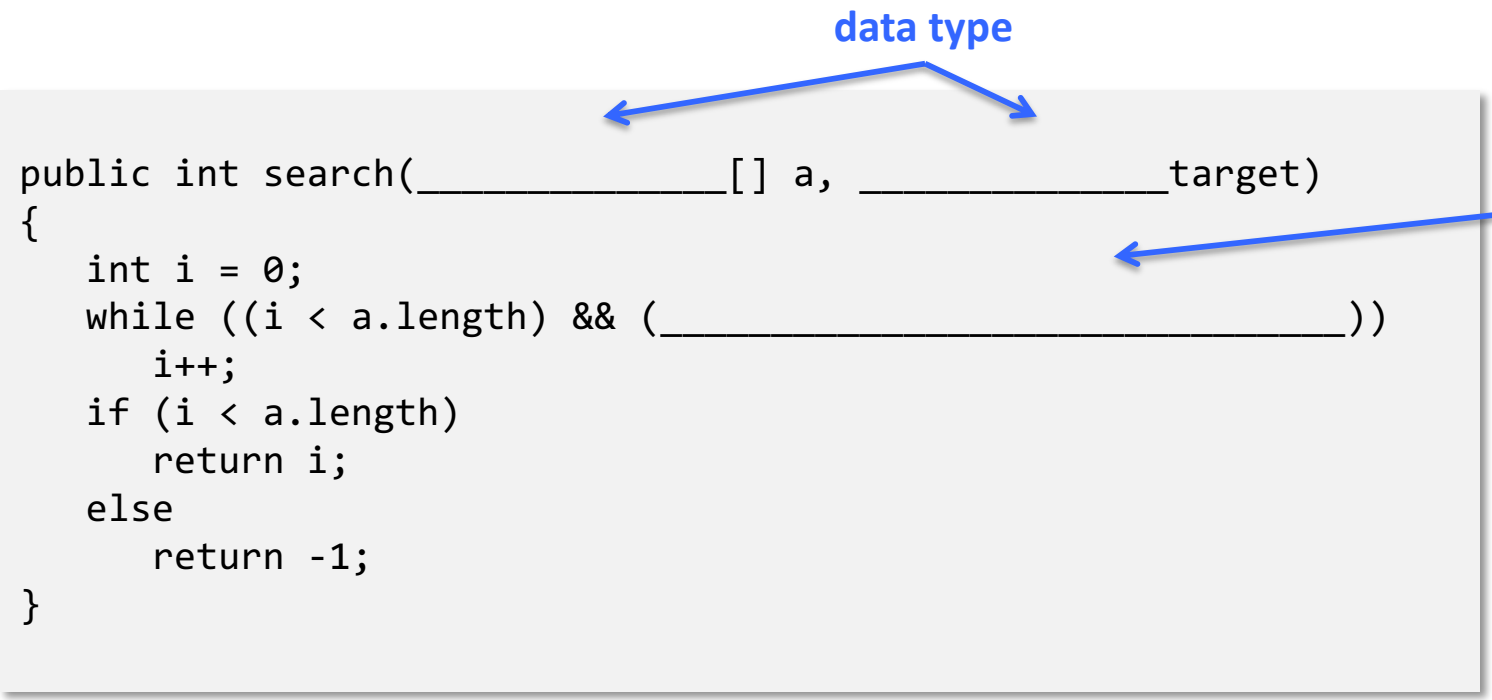
The natural ordering for a class C is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

It is strongly recommended (though not required) that natural orderings be consistent with equals.

Total order:

http://en.wikipedia.org/wiki/Total_order

General data types



General data types

data type

```
public int search(__Comparable_[] a, __Comparable_target)
{
   int i = 0;
   while ((i < a.length) && (a[i].compareTo(target) != 0))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

It's the responsibility of the underlying data class to provide an appropriate implementation of the compareTo method.

How one object of this type can be compared to another

The compareTo contract

http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

```
int compareTo(T o)
```

Compares this object with the specified object for order. **Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.**

The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation sgn(expression) designates the mathematical **signum function**, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

sgn: http://en.wikipedia.org/wiki/Sign_function

Participation

Q: Consider the following code.

```
String s1 = “A”;
String s2 = “F”;

int cmp = s1.compareTo(s2);
```

What value should we expect the variable cmp to contain?

A. 0
B. -1
C. 1
D. some negative integer
E. some positive integer

Client side

```
public void testSearchString() {
    String[] a = {"2", "4", "6", "8", "10"};
    String target = "8";
    int expected = 3;
    int actual = search(a, target);

    assert actual == expected;
 }
```

This will work – String implements Comparable.

Client side

```
public static void testSearchInteger() {
     Integer[] a = {2, 4, 6, 8, 10};
     Integer target = 8;
     int expected = 3;
     int actual = search(a, target);

     assert actual == expected;
  }
```

This will work – Integer implements Comparable.

Client side

```
public static void testSearchBook() {
     Book[] a = {new Book("author1", "title1", 123),
           new Book("author2", "title2", 456),
           new Book("author3", "title3", 789),
           new Book("author4", "title4", 123),
           new Book("author5", "title5", 456)};
     Book target = new Book("author4", "title4", 999);
     int expected = 3;
     int actual = search(a, target);

     assert actual == expected;
  }
```

This will NOT COMPILE unless Book implements Comparable.

A compareTo recipe

```
public class Book implements Comparable {
   private String author;
   private String title;
   private int pages;
   ...

public int compareTo(Object obj) {
    Book that = (Book) obj;

    int cmp = this.title.compareTo(that.title);

    if (cmp == 0) {
       cmp = this.author.compareTo(that.author);
    }

    return cmp;
}
```

Must implement the interface

Preferably not Object; stay tuned …

Cast from Object, exceptions possible

From most significant field to least significant field, make comparisons until order (<, >, ==) can be determined.

As recommended, this is consistent with Book's equals method.

General data types

```
public int search(Comparable[] a, Comparable target) {
   int i = 0;
   while ((i < a.length) && (a[i].compareTo(target) != 0))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Pro:
General and appropriate for algorithms needing a total order

Con:
Only appropriate for comparison-based algorithms
Can't guarantee type-safety as-is

Instances of distinct, incompatible classes that implement the Comparable interface are legal arguments to this method.

```
Comparable[] a = {"red", new Book("A", "T", 123), new Integer(15)};

search(a, new Double(3.14));
```

Type safety

Type safety

Type safety refers to the extent to which typing errors are prevented.

Java attempts to catch typing errors at compile time to prevent them from occurring while the program is running.

All of the following typing errors are caught by the compiler.

```
int i = 3.14;
```
✗

```
String s = "2";
```
✓

```
int sum = s + 5;
```
✗

```
public void foo(Number a, Double b, Integer c) { . . . }
```
✓

```
foo(3.14, 3.14, 3.14);
```
✗

A Java program is considered **type safe** if there are no definite or potential type errors identified by the compiler.

Type safety

```
public class ArrayLib1 {

   public static int search(Object[] a, Object target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target)))
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib1.search(sa, "8");
   }
}
```

```
% javac ArrayLib1.java
%
```

type safe

Type safety

```
public class ArrayLib2 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib2.search(sa, "8");
   }
}
```

```
% javac ArrayLib2.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
%
```

NOT type safe

Type safety

```
public class ArrayLib2 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib2.search(sa, "8");
   }
}
```

```
% java ArrayLib2
%
```

NOT type safe, but no runtime errors in this case.

Type safety

```
public class ArrayLib2 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib2.search(sa, new Double(3.14));
   }
}
```

```
% java ArrayLib2
Exception in thread "main" java.lang.ClassCastException: java.lang.Double
cannot be cast to java.lang.String
%
```

This example generates a runtime error because the code is not type safe.

Type safety

```
public class ArrayLib3 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib3.search(sa, new Double(3.14));
   }
}
```

```
% javac ArrayLib3.java
%
```

type safe

Type safety

http://download.oracle.com/javase/tutorial/extra/generics/fineprint.html

> *"In particular, the language is designed to guarantee that if your entire application has been compiled* ***without unchecked warnings*** *using javac -source 1.5, it is* ***type safe****."*

An unchecked warning is issued by a Java compiler to indicate that not enough type information is available for the compiler to ensure that no unexpected type error will occur at runtime.

Participation

Q: What happens when we try to compile and run the following code?

```
public static int search(Object[] a, Object target) {
   . . .
}
. . .
Object[] a = {"red", new Book("A", "T", 123), new Integer(15)};

search(a, new Double(3.14));
```

A. Generates a compile-time error
B. Compiles correctly but generates a runtime error
C. Compiles correctly and runs without error

Participation

Q: What happens when we try to compile and run the following code?

```
public static int search(Comparable[] a, Comparable target) {
   . . .
}
. . .
Object[] a = {"red", new Book("A", "T", 123), new Integer(15)};

search(a, new Double(3.14));
```

A. Generates a compile-time error
B. Compiles correctly but generates a runtime error
C. Compiles correctly and runs without error

Participation

Q: What happens when we try to compile and run the following code?

```
public static int search(Comparable[] a, Comparable target) {
   . . . // uses compareTo()
}
. . .
Comparable[] a = {"red", new Book("A", "T", 123), new Integer(15)};

search(a, new Double(3.14));
```

A. Generates a compile-time error
B. Compiles correctly but generates a runtime error
C. Compiles correctly and runs without error

Generics

Generics

Java allows a **type variable** to be used in place of a specific type name.

A type variable can be used to *parameterize* a class, interface, or method with respect to the types involved.

This **generic typing** allow classes, interfaces, and methods to deal with objects of different types at runtime while maintaining compile-time **type safety**.

```
public <T> int search(T[] a, T target)
{
   int i = 0;
   while ((i < a.length) && (!a[i].equals(target)))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Parameterizing the **method**

Client side

```
public class ArrayLib {
   public static <T> int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      }
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

In a client …

```
String[] sarray = {"2","4","6","8","10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

The client can supply a value for the type variable when the method is called:

```
ArrayLib.<String>search(sarray, “8”)
```

Tells the compiler to associate the actual type String with the type variable T.

Client side

```
public class ArrayLib {
   public static <T> int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      }
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

In a client …

```
String[] sarray = {"2","4","6","8","10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

The client can leave the type variable unspecified:

```
ArrayLib.search(sarray, “8”)
```

This essentially opts-out of the generic type system and is poor practice.

Client side

```
public class ArrayLib {
   public static <T> int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      }
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

In a client …

```
String[] sarray = {"2","4","6","8","10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

Sample calls:

```
ArrayLib.<String>search(sarray, “8”)
```

```
ArrayLib.<String>search(sarray, 8)
```

```
ArrayLib.search(sarray, 8)
```

(But don’t do this.)

Client side

```
public class ArrayLib {
   public static <T> int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      }
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

In a client …

```
String[] sarray = {"2","4","6","8","10"};
Integer[] iarray = {2, 4, 6, 8, 10};
Number[] narray = {2, 4, 6, 8, 10};
```

Sample calls:

```
ArrayLib.<Integer>search(iarray, 8)
```

```
ArrayLib.<Integer>search(narray, 8)
```

```
ArrayLib.<Number>search(iarray, 8)
```

Type parameters

```
public class ArrayLib<T> {
   public int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (!a[i].equals(target))) {
         i++;
      }
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

Parameterizing the **class**

Since T is a type variable, the ArrayLib class must be instantiated in order to bind T to a concrete type.

So, the search method is no longer static.

Client side:

```
ArrayLib<String>  alstr  = new ArrayLib<String>();
ArrayLib<Integer> alint  = new ArrayLib<Integer>();
ArrayLib<Book>    albook = new ArrayLib<Book>();
```

Type parameters

Many classes, interfaces, and methods in Java have type parameters . . .

```
public interface Comparable<T> {
   . . .
   int compareTo(T o) { . . . }
}
```

```
public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable {
  . . .
  boolean add(E e) {  . . . }
  . . .
}
```

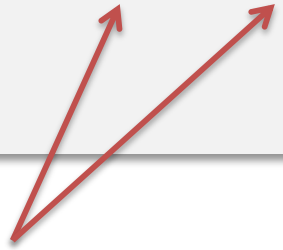
. . . and you should always bind the type parameters when using them.

Terms associated with generics in Java

Usually the terms “generics”, “generic type”, etc. are used colloquially and the real meaning is obscured. The following terms are taken from the Java Language Specification (JLS) Java SE 8 Edition (2014-03-03). http://docs.oracle.com/javase/specs/jls/se8

A **type variable** is an unqualified identifier used as a type in class, interface, method, and constructor bodies.

```
public class ArrayLib<T> {
  public int search(T[] a, T target) {
    . . .
  }
}
```



T is a type variable in the ArrayLib class.

Terms associated with generics in Java

A class, interface, or method is **generic** if it declares one or more type variables.

```
public class ArrayLib<T> { . . . }
```

ArrayLib is a generic class.

```
public interface Comparable<T> { . . . }
```

Comparable is a generic interface.

Generic classes and interfaces are called **generic types**.

Terms associated with generics in Java

A type variable is introduced by the declaration of a **type parameter** of a generic class, interface, method, or constructor


```
public class ArrayLib<T> {
  public int search(T[] a, T target) {
    . . .
  }
}
```

A generic type can have more than one type parameter.

```
public class HashMap<K, V> {
  . . .
}
```

Terms associated with generics in Java

A generic class or interface defines a set of **parameterized types**. A parameterized type is of the form $C<T_1, T_2, \ldots T_N>$ where C is the name of a generic type and $<T_1, T_2, \ldots T_N>$ is a list of **type arguments** that denote a particular *parameterization* of the generic type.

```
ArrayLib<String>
ArrayLib<Integer>
ArrayLib<Book>
```

Three parameterized types that represent three distinct parameterizations of the generic type ArrayLib. String, Integer, and Book are the three different type arguments used.

Parameterized types exist only at compile-time. All type arguments are eliminated ("erased") through a process known as **type erasure** and do not exist at runtime.

```
ArrayLib<String>  alstr = new ArrayLib<String>();
ArrayLib<Integer> alint = new ArrayLib<Integer>();
```

The declared type of alstr is ArrayLib<String>. The runtime type of alstr is ArrayLib.

The declared type of alint is ArrayLib<Integer>. The runtime type of alint is ArrayLib.

Terms associated with generics in Java

Every generic type defines one *raw type*. A **raw type** is a generic type name without parameterization; that is, a raw type is the *erasure* of a generic type.

Example: ArrayLib is the raw type of ArrayLib<T>.

You should not declare variables or parameters of raw types in the code that you write.

```
ArrayLib al = new ArrayLib();   // poor practice
```

If you use raw types you lose all the safety and expressiveness of generic types.

From the JLS (4.8): "*The use of raw types is allowed only as a concession to compatibility of legacy code. The use of raw types in code written after the introduction of generics into the Java programming language is strongly discouraged. It is possible that future versions of the Java programming language will disallow the use of raw types.*"

The use of raw types is a common source of unchecked warnings in assignment code.

Type safety

```
public class ArrayLib2 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib2.search(sa, "8");
   }
}
```

Comparable is the raw type of Comparable<T>

With the raw type, there is no way for the compiler to know if compareTo will throw a ClassCastException or not.

```
% javac ArrayLib2.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
%
```

NOT type safe

Type safety

```
% javac ArrayLib2.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
%
```

Xlint is is a “nonstandard” option of the javac compiler that provides more detailed information on warnings that the compiler produces. You can choose all warnings (-Xlint:all) or specific warnings (e.g., -Xlint:unchecked).

http://docs.oracle.com/javase/8/docs/technotes/tools/unix/javac.html

```
% javac -Xlint:unchecked ArrayLib2.java
ArrayLib2.java:5: warning: [unchecked] unchecked call to compareTo(T) as a
member of the raw type Comparable
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
                                                ^
  where T is a type-variable:
    T extends Object declared in interface Comparable
1 warning
%
```

Terms associated with generics in Java

A type variable can have a **bound** specified in its declaration using the "extends type" notation.

```
public class ArrayLib<T extends Number> {
  public int search(T[] a, T target) {
    . . .
  }
}
```

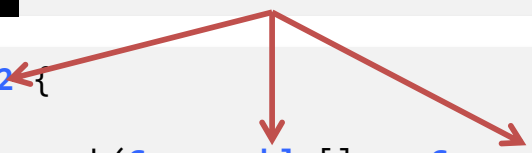
Number is the **upper bound** of T, so the only legal type arguments are Number and subclasses of Number.

```
ArrayLib<Integer> ali = new ArrayLib<Integer>();
ArrayLib<Double>  ald = new ArrayLib<Double>();
ArrayLib<String>  als = new ArrayLib<String>();
```

The specified bound can be any type in Java.

Type safety

Use a generic type variable with Comparable as its upper bound.



```
public class ArrayLib2 {

   public static int search(Comparable[] a, Comparable target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0)) {
         i++;
      if (i < a.length)  return  i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      int i = ArrayLib2.search(sa, "8");
   }
}
```

```
% javac ArrayLib2.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
%
```

NOT type safe

Type safety

```
public class ArrayLib5<T extends Comparable> {

   public int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0))
         i++;
      if (i < a.length)  return i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      ArrayLib5<String> als = new ArrayLib5<String>();
      als.search(sa, “8”);
   }
}
```

```
% javac ArrayLib5.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with –Xlint:unchecked for details.
%
```

NOT type safe

Type safety

Raw type!

```
public class ArrayLib5<T extends Comparable> {

   public int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0))
         i++;
      if (i < a.length)  return i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      ArrayLib5<String> als = new ArrayLib5<String>();
      als.search(sa, “8”);
   }
}
```

```
% javac ArrayLib5.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with –Xlint:unchecked for details.
%
```

NOT type safe

Type safety

Raw type!

```
public class ArrayLib5<T extends Comparable> {

   public int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0))
         i++;
      if (i < a.length)  return i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      ArrayLib5<Comparable> alc = new ArrayLib5<Comparable>();
      alc.search(sa, new Double(3.14));
   }
}
```

Compiler can't ensure that the types in a and target are ***mutually*** comparable.

```
% javac ArrayLib5.java
Note: ArrayLib2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
%
```

NOT type safe, and this example will cause a runtime error.

Comparable

Comparable is a generic type.

```
public interface Comparable<T> { . . . }
```

When we implement Comparable, we should **bind its type parameter**.

```
public class Book implements Comparable<Book> {
    . . .

    public int compareTo(Book b) {
       // no casting!
    }
}
```

Type safety

```
public class ArrayLib6<T extends Comparable<T>> {

   public int search(T[] a, T target) {
      int i = 0;
      while ((i < a.length) && (a[i].compareTo(target) != 0))
         i++;
      if (i < a.length)  return i;
      else               return -1;
   }

   public static void main(String[] args) {
      String[] sa = {"2", "4", "6", "8", "10"};
      ArrayLib6<String> als = new ArrayLib6<String>();
      als.search(sa, “8”);   }
}
```

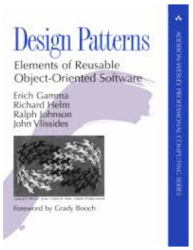
This will only accept a type that can be compared to itself.

```
% javac ArrayLib6.java
%
```

type safe

Generalizing behavior – Comparator

Strategy pattern



Design patterns – "Gang of Four" book
Erich Gamma, Richard Helm, Ralph Johnson, John Vissides

Strategy pattern: Allow an algorithm's behavior to selected at runtime.

```
public int search(T[] a, T target)
{
   int i = 0;
   while ((i < a.length) && (_____________________________))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

What if we wanted to allow the client – not the data class – to specify exactly how this comparison is to be done?

Comparator interface

http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

```
@FunctionalInterface
public interface Comparator<T>
```

A comparison function, which **imposes a total ordering** on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that *don't have a natural ordering*.

The ordering imposed by a comparator c on a set of elements S is said to be consistent with equals if and only if c.compare(e1, e2)==0 has the same boolean value as e1.equals(e2) for every e1 and e2 in S.

Comparator interface

http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

```
int compare(T o1, T o2)
```

Compares its two arguments for order. **Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.**

In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y. (This implies that compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.)

The implementor must also ensure that the relation is transitive: ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementor must ensure that compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z.

It is generally the case, but not strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Using Comparators

```
public class ArrayLib {

   public static <T> int search(T[] a, T target, Comparator<T> c) {
      int i = 0;
      while ((i < a.length) && (c.compare(a[i], target) != 0))
         i++;
      if (i < a.length)
         return i;
      else
         return -1;
   }
}
```

This allows the client to search through the same data using different strategies for the comparison.

For example, searching through an array of Books for author, then again for title.

Client side

```
Book[] a = { new Book("Author A", "Title Z", 456),
             new Book("Author B", "Title Y", 123),
             new Book("Author C", "Title X", 789)};

Book target = new Book("Author B", "Title Z", 543);
```

```
class CompareBooksByTitle implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
       return b1.getTitle().compareTo(b2.getTitle());
    }
}
```

Note: Java 8 allows lambda expressions, which provide a nice alternative to the class definition/instantiation pattern shown here.

Calling the generic method with this comparator:

```
ArrayLib.<Book>search(a, target, new CompareBooksByTitle());
```

Client side

We can define as many Book comparators as we like:

```
class CompareBooksByTitle implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
       return b1.getTitle().compareTo(b2.getTitle());
    }
 }
```

```
class CompareBooksByAuthor implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
       return b1.getAuthor().compareTo(b2.getAuthor());
    }
 }
```

```
class CompareBooksByLength implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
       return b1.pages – b2.pages;
    }
 }
```

Client side

```
public class ComparatorClient {
   public static void main(String[] args) {
      Book[] a = {
            new Book("Author A", "Title Z", 456),
            new Book("Author B", "Title Y", 123),
            new Book("Author C", "Title X", 789)
            };
      Book target = new Book("Author B", "Title Z", 543);

      CompareBooksByAuthor authorComparator = new CompareBooksByAuthor();
      CompareBooksByTitle titleComparator = new CompareBooksByTitle();
      CompareBooksByLength lengthComparator = new CompareBooksByLength();

      ArrayLib<Book> alb = new ArrayLib<Book>();

      System.out.println(alb.search(a, target, authorComparator));
      System.out.println(alb.search(a, target, titleComparator));
      System.out.println(alb.search(a, target, lengthComparator));

   }
}
```

```
% java ComparatorClient
1
0
-1
```

Arrays.sort() with comparators

```
import java.util.Arrays;
import java.util.Comparator;

public class ComparatorSortExample {
   public static void main(String[] args) {
      Book[] a = {
            new Book("Author A", "Title Z", 456),
            new Book("Author B", "Title Y", 123),
            new Book("Author C", "Title X", 789)
            };

      CompareBooksByAuthor authorComparator = new CompareBooksByAuthor();
      CompareBooksByTitle titleComparator = new CompareBooksByTitle();
      CompareBooksByLength lengthComparator = new CompareBooksByLength();

      print(a);
      Arrays.<Book>sort(a, titleComparator);   print(a);
      Arrays.<Book>sort(a, lengthComparator);  print(a);
      Arrays.<Book>sort(a, authorComparator);  print(a);
   }
}
```

Arrays.sort() with comparators

```
import java.util.Arrays;
import java.util.Comparator;

public class ComparatorSortExample {
   public static void main(String[] args) {
      Book[] a = {
            new Book("Author A", "Title Z", 456),
            new Book("Author B", "Title Y", 123),
            new Book("Author C", "Title X", 789)
            };

      CompareBooksByAuthor authorComparator = new CompareBooksByAut
      CompareBooksByTitle titleComparator = new CompareBooksByTitle
      CompareBooksByLength lengthComparator = new CompareBooksByLen

      print(a);
      Arrays.<Book>sort(a, titleComparator);   print(a);
      Arrays.<Book>sort(a, lengthComparator);  print(a);
      Arrays.<Book>sort(a, authorComparator);  print(a);
   }
}
```

```
% java ComparatorSortExample
Author A   Title Z   456
Author B   Title Y   123
Author C   Title X   789

Author C   Title X   789
Author B   Title Y   123
Author A   Title Z   456

Author B   Title Y   123
Author A   Title Z   456
Author C   Title X   789

Author A   Title Z   456
Author B   Title Y   123
Author C   Title X   789
```

Generalizing structure – Collections

Making code general

```
public int search(int[] a, int target) {
   int i = 0;
   while ((i < a.length) && (a[i] != target))
      i++;
   if (i < a.length)
      return i;
   else
      return -1;
}
```

Two basic issues to address in making this code more general:

The data type being used

How the "list" is represented

What we'll do: Choose a very general data type but retain type-safety.

Select a representation for the "list" that's less concrete and more flexible.

Employ general programming idioms that tend to make code more reusable.

Using Collections

http://download.oracle.com/javase/6/docs/technotes/guides/collections/index.html

"The collections framework is a unified architecture for representing and manipulating **collections**, allowing them to be manipulated **independently of the details of their representation**. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and **fosters software reuse**. The framework is based on fourteen collection interfaces. It includes implementations of these interfaces, and algorithms to manipulate them."

Probably the closest match to our problem: **java.util.List**

```
public interface List<E> extends Collection<E> { ...
```

Array feature used	Replace with List method
`a.length`	`int size()`
`a[i]`	`E get(int index)`

And let's just ignore the indexOf() method ...

Searching a List

```
public <T> int search(___List<T>_____ a, __T__ target)
{
   int i = 0;
   while ((__i < a.size()_____) && (__!a.get(i).equals(target)_____))
      i++;
   if (___i < a.size()_____)
      return i;
   else
      return -1;
}
```

Remember that List is an **interface**, not a class.

This code doesn't require any particular class to be used as long as it implements the List interface.

Client side

```
public class ListSearchClient {

   public static void main(String[] args) {

      Book[] a = {
            new Book("Author A", "Title Z", 456),
            new Book("Author B", "Title Y", 123),
            new Book("Author C", "Title X", 789)
            };

      List<Book> bla = Arrays.asList(a);

      Book b = new Book("Author B", "Title Y", 987);

      System.out.println(SearchLib.<Book>search(bla, b));
   }
}
```

Client side

```
public class ListSearchClient {

   public static void main(String[] args) {

      List<Book> bll = new LinkedList<Book>();

      bll.add(new Book("Author A", "Title Z", 456));
      bll.add(new Book("Author B", "Title Y", 123));
      bll.add(new Book("Author C", "Title X", 789));

      Book b = new Book("Author B", "Title Y", 987);

      System.out.println(SearchLib.<Book>search(bll, b));
   }
}
```

Client side

```
public class ListSearchClient {

   public static void main(String[] args) {

      List<String> sla = new ArrayList<String>();

      sla.add("red"); sla.add("orange"); sla.add("yellow");
      sla.add("green"); sla.add("blue"); sla.add("indigo");
      sla.add("violet");

      System.out.println(SearchLib.<String>search(sla, “magenta”));
   }
}
```

Client side

```
public class ListSearchClient {

   public static void main(String[] args) {

      List<Integer> vli = new Vector<Integer>();

      vli.add(2); vli.add(4); vli.add(6); vli.add(8); vli.add(10);

      System.out.println(SearchLib.<Integer>search(vli, 8));
   }
}
```

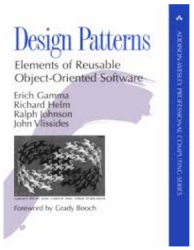
Searching a List

```
public <T> int search(List<T> a, T target)
{
   int i = 0;
   while ((i < a.size()) && (!a.get(i).equals(target)))
      i++;
   if (i < a.size())
      return i;
   else
      return -1;
}
```

All these different collections with different data types are handled by this one method.

Generalizing behavior – Iterator

Iterator pattern



Design patterns – "Gang of Four" book
Erich Gamma, Richard Helm, Ralph Johnson, John Vissides

Iterator pattern: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

"In computer science, an **iterator** is an object that allows a programmer to **traverse** through all the elements of a collection, **regardless of its specific implementation**."

"**Iterators** are **central to generic programming** because they are an interface between containers and algorithms. [Iterators] **make it possible to write a generic algorithm that operates on many different kinds of containers**, *even containers as different as a vector and a doubly linked list*."

Iterator interface

http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

```
public interface Iterator<E>
```

`boolean hasNext()`

Returns true if the iteration has more elements.

`E next()`

Returns the next element in the iteration. Throws NoSuchElementException if the iteration has no more elements

`void remove()`

Removes from the underlying collection the last element returned by this iterator (**optional operation**). Throws UnsupportedOperationException if the remove operation is not supported by this iterator

Iterator pattern

```
public <T> int search(List<T> a, T target)
{
   Iterator<T> itr = a.iterator();
   int i = 0;
   while ((itr.hasNext()) && (!itr.next().equals(target)))
      i++;
   if (i < a.size())
      return i;
   else
      return -1;
}
```

The iterator method is part of the List interface

The next method returns the next element in the iteration and advances the cursor. This is a common source of errors.

Iterable interface

```
public <T> int search(List<T> a, T target) {
   int i = 0;
   for (T element : a) {
      if (element.equals(target)) {
         return i;
      }
      i++;
   }
   return -1;
}
```

The Iterable interface is a way for a class to state that it provides an iterator() method to create an iterator on its elements.

This allows objects of this class to be the target of the “for-each” loop in Java.

Benefits of an iterator

Generality Allows traversals regardless of implementation,

including the methods that are available.

Not every collection will have a "get by index" method.

Efficiency Depending on how the collection is implemented, the iterator could provide much faster access than the get() method in a loop.

List Implementation	Search with get()	Search with iterator
java.util.ArrayList	7 milliseconds	14 milliseconds
java.util.LinkedList	**4076** milliseconds	21 milliseconds

Average time over four runs. List size = 100,000