Chapter 15 -The Java Collections Framework

Backtracking - Maze Example

- Start, at position (3, 4).
- There are four possible paths. We push them all on a stack (1).
- We pop off the topmost one, traveling north from (3, 4).
- Following this path leads to position (1, 4).

  We now push two choices on the stack, going west or east (2).

  Both of them lead to dead ends (3) (4).

- Now we pop off the path from (3,4) going east.

  That too is a dead end (5).

- Next is the path from (3, 4) going south.
- Comes to an intersection at (5, 4).

  Both choices are pushed on the stack (6).

  They both lead to dead ends (7) (8).

- Finally, the path from (3, 4) going west leads to an exit (9).

Backtracking

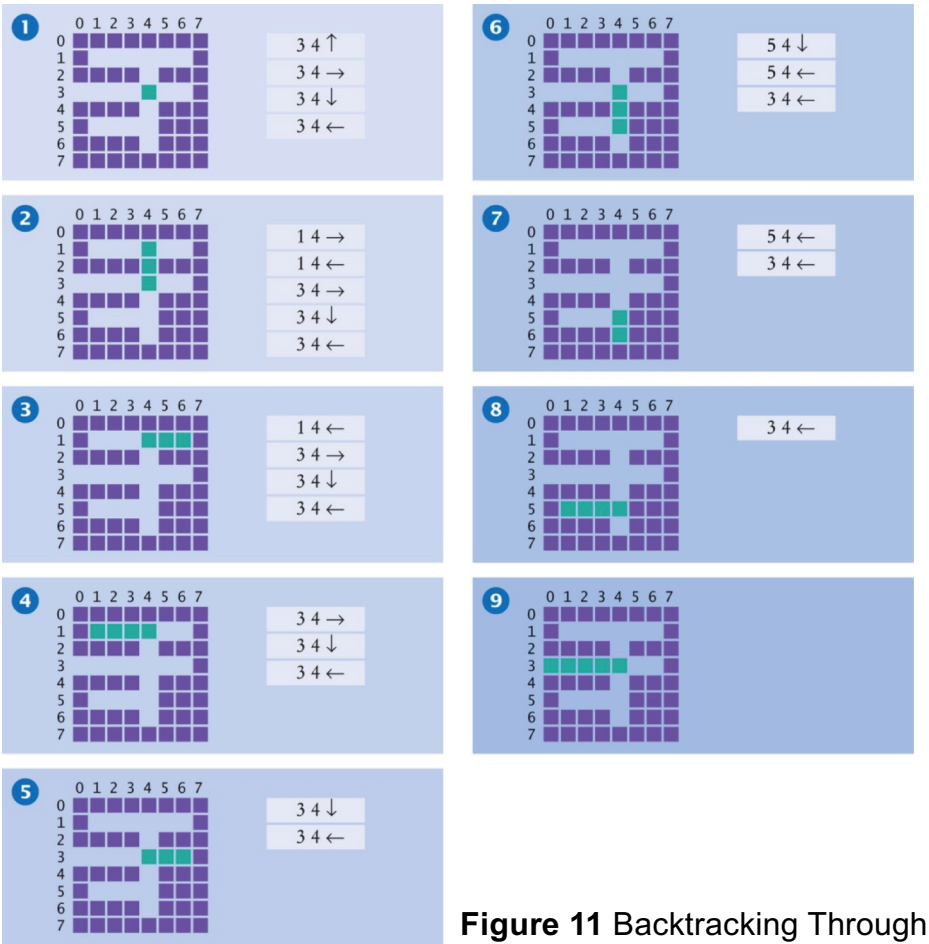


Figure 11 Backtracking Through a Maze

Backtracking

- Algorithm:

```
Push all paths from the point on which you are standing on a stack.
While the stack is not empty
  Pop a path from the stack.
  Follow the path until you reach an exit, intersection, or dead end.
  If you found an exit
    Congratulations!
Else if you found an intersection
  Push all paths meeting at the intersection, except the current one, onto the  stack.
```

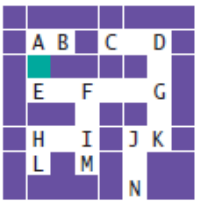
- This works if there are no cycles in the maze.

  You never circle back to a previously visited intersection

- You could use a queue instead of a stack.

Self Check 15.30

Consider the following simple maze. Assuming that we start at the marked point and push paths in the order West, South, East, North, in which order are the lettered points visited, using the algorithm of Section 15.6.4?



Answer: A B E F G D C K J N

Chapter 18 - Generic Classes

Generic Classes and Type Parameters

- **Generic programming:** creation of programming constructs that can be used with many different types.
  - In Java, achieved with type parameters or with inheritance
  - Type parameter example: Java's `ArrayList` (e.g. `ArrayList<String>`)
  - Inheritance example: `LinkedList` implemented in Section 16.1 can store objects of any class
- **Generic class:** has one or more type parameters.
- A type parameter for `ArrayList` denotes the element type:

```
public void add(E element)
public E get(int index)
```

Type Parameter

- Can be instantiated with class or interface type:

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

- Cannot use a primitive type as a type parameter:

```
ArrayList<double> // Wrong!
```

- Use corresponding wrapper class instead:

```
ArrayList<Double>
```

Type Parameters

- Supplied type replaces type variable in class interface.
- Example: `add` in `ArrayList<BankAccount>` has type variable `E` replaced with `BankAccount`:

```
public void add(BankAccount element)
```

- Contrast with `LinkedList.add` from Chapter 16:

```
public void add(Object element)
```

Type Parameters Increase Safety

- Type parameters make generic code safer and easier to read:
  - Impossible to add a `String` into an `ArrayList<BankAccount>`
  - Can add a `String` into a non-generic `LinkedList` intended to hold bank accounts

```
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();
// Should hold BankAccount objects
LinkedList accounts2 = new LinkedList();
accounts1.add("my savings"); // Compile-time error
accounts2.add("my savings"); // Not detected at compile time
. . .
// Run-time error
BankAccount account = (BankAccount) accounts2.getFirst();
```

Self Check 18.3

Does the following code contain an error? If so, is it a compile-time or run-time error?

```
ArrayList<Integer> a = new ArrayList<>();
String s = a.get(0);
```

Answer: This is a compile-time error. You cannot assign the `Integer` expression `a.get(0)` to a string.

Self Check 18.5

Does the following code contain an error? If so, is it a compile-time or run-time error?

```
LinkedList a = new LinkedList();
a.addFirst("3.14");
double x = (Double) a.removeFirst();
```

Answer: This is a run-time error. `a.removeFirst()` yields a `String` that cannot be converted into a `Double`.

Remedy: Call `a.addFirst(3.14);`

Implementing Generic Classes

- Example: simple generic class that stores *pairs* of arbitrary objects such as:

```
Pair<String, Integer> result
      = new Pair<>("Harry Hacker", 1729);
```

- Methods `getFirst` and `getSecond` retrieve first and second values of pair:

```
String name = result.getFirst();
Integer number = result.getSecond();
```

- Generic `Pair` class requires two type parameters, one for each element type enclosed in angle brackets:

```
public class Pair<T, S>
```

Class Pair

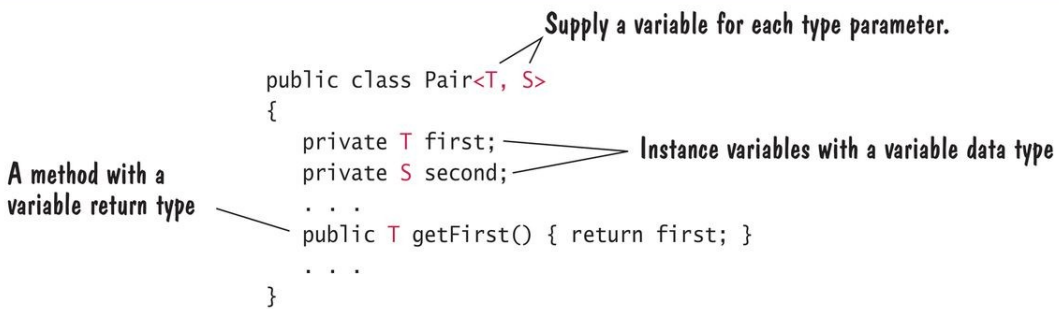
```
public class Pair<T, S>
{
   private T first;
   private S second;

   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
}
```

Syntax 18.1 Declaring a Generic Class

Syntax

modifier `class` *GenericClassName*`<`*TypeVariable*$_1$`,` *TypeVariable*$_2$`, . . .>`
`{`
instance variables
constructors
methods
`}`



Self Check 18.8

What is the difference between an `ArrayList<Pair<String, Integer>>` and a `Pair<ArrayList<String>, Integer>`?

Answer: An `ArrayList<Pair<String, Integer>>` contains multiple pairs, for example `[(Tom, 1), (Harry, 3)]`. A `Pair<ArrayList<String>, Integer>` contains a list of strings and a single integer, such as `([Tom, Harry], 1)`.

Self Check 18.9

Write a method `roots` with a `Double` parameter variable `x` that returns both the positive and negative square root of `x` if `x ≥ 0` or `null` otherwise.

Answer:

```
public static Pair<Double, Double> roots(Double x)
{
   if (x >= 0)
   {
      double r = Math.sqrt(x);
      return new Pair<Double, Double>(r, -r);
   }
   else { return null; }
}
```

Self Check 18.10

How would you implement a class `Triple` that collects three values of arbitrary types?

Answer: You have three type parameters: `Triple<T, S, U>`. Add an instance variable `U` third, a constructor argument for initializing it, and a method `U getThird()` for returning it.

Generic Methods

- **Generic method:** method with a type parameter.
- Can be declared inside non-generic class.
- Example: Declare a method that can print an array of any type:

```
public class ArrayUtil
{
   /**
      Prints all elements in an array.
      @param a the array to print
   */
   public <T> static void print(T[] a)
   {
      . . .
   }
   . . .
}
```

Generic Methods

- Often easier to see how to implement a generic method by starting with a concrete example.
- Example: print the elements in an array of *strings*:

```
public class ArrayUtil
{
   public static void print(String[] a)
   {
      for (String e : a)
      {
         System.out.print(e + " ");
      }
      System.out.println();
   }
   . . .
}
```

Generic Methods

- In order to make the method into a generic method:

  Replace `String` with a type parameter, say `E`, to denote the element type.

  Add the type parameters between the method's modifiers and return type.

```
public static <E> void print(E[] a)
{
   for (E e : a)
   {
      System.out.print(e + " ");
   }
   System.out.println();
}
```

Generic Methods

- When calling a generic method, you need not instantiate the type variables:

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- The compiler deduces that `E` is `Rectangle`.
- You can also define generic methods that are not static.
- You can even have generic methods in generic classes.
- Cannot replace type variables with primitive types.

  Example: cannot use the generic `print` method to print an array of type `int[]`

Self Check 18.11

Exactly what does the generic `print` method print when you pass an array of `BankAccount` objects containing two bank accounts with zero balances?

Answer: The output depends on the definition of the `toString` method in the `BankAccount` class.

Constraining Type Variables

You can place restrictions on the type parameters of generic classes and methods.

Constraining Type Variables

- Type variables can be constrained with bounds.
- A generic method, `average`, needs to be able to measure the objects.
- Let us take for example the Measurable interface:

```
public interface Measurable
{
   double getMeasure();
}
```

- We can constrain the type of the elements to those that implement the `Measurable` type:

```
public static <E extends Measurable> double average(ArrayList<E>
objects)
```

- This means, “E or one of its superclasses extends or implements Measurable”.

We say that `E` is a subtype of the `Measurable` type.

Constraining Type Variables

- Completed `average` method:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
{
   if (objects.size() == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.size();
}
```

- In the call `obj.getMeasure()`

  It is legal to apply the `getMeasure` method to `obj`.

  `obj` has type `E`, and `E` is a subtype of `Measurable`.

Constraining Type Variables - Comparable Interface

- `Comparable` interface is a generic type.
- The type parameter specifies the type of the parameter variable of the `compareTo` method:

```
public interface Comparable<T>
{
   int compareTo(T other);
}
```

- `String` class implements `Comparable<String>`

  A String can be compared to other `String`.

  But not with objects of a different class.

Constraining Type Variables - Comparable Interface

- When writing a generic method `min` to find the smallest element in an array list,

  Require that type parameter `E` implements `Comparable<E>`

```
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
{  E smallest = objects.get(0);
  for (int i = 1; i < objects.size(); i++)
  { E obj = objects.get(i);
    if (obj.compareTo(smallest) < 0)
    { smallest = obj;
    }
  }
  return smallest;
}
```

- Because of the type constraint, the class of `obj` must have a method of this form:

```
int compareTo(E other)
```

  So the the following call is valid:

```
obj.compareTo(smallest)
```

Constraining Type Variables

- Very occasionally, you need to supply two or more type bounds:

```
<E extends Comparable<E> & Cloneable>
```

- `extends`, when applied to type parameters, actually means “extends or implements.”
- The bounds can be either classes or interfaces.
- Type parameters can be replaced with a class or interface type.

Self Check 18.16

How would you constrain the type parameter for a generic `BinarySearchTree` class?

Answer:

```
public class BinarySearchTree<E extends Comparable<E>>
```

Self Check 18.20

How would you implement the generic `average` method for arrays?

Answer:

```
public static <E extends Measurable> double average(E[] objects)
{ if (objects.length == 0) { return 0; }
  double sum = 0;
  for (E obj : objects)
  { sum = sum + obj.getMeasure();
  }
  return sum / objects.length;
}
```

Self Check 18.21

Is it necessary to use a generic average method for arrays of measurable objects?

Answer: No. You can define

```
public static double average(Measurable[] objects)
{
   if (objects.length == 0) { return 0; }
   double sum = 0;
   for (Measurable obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
}
```

For example, if `BankAccount` implements `Measurable`, a `BankAccount[]` array is convertible to a `Measurable[]` array. Here, the return type is `double`, and there is no need for using generic types.

Genericity and Inheritance

- One can not assign a subclass list to a superclass list.

  `ArrayList<SavingsAccount>` is not a subclass of `ArrayList<BankAccount>`.

  Even though `SavingsAccount` is a subclass of `BankAccount`

```
ArrayList<SavingsAccount> savingsAccounts = new
ArrayList<SavingsAccount>();
// Not legal - compile-time error
ArrayList<BankAccount> bankAccounts = savingsAccounts;
```

- However, you can do the equivalent thing with arrays:

```
SavingsAccount[] savingsAccounts = new SavingsAccount[10];
BankAccount bankAccounts = savingsAccounts; // Legal
```