ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: exceptions and genericity

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Exceptions

Exceptions

Exceptions are objects

- Raise with a **throw** **ExceptionObject** instruction

```
throw new AnExceptionClass("ErrorInfo");
```

- Checked exceptions

  Declared in method signature:

```
public void foo() throws SomeCheckedException
```

  Must be handled explicitly

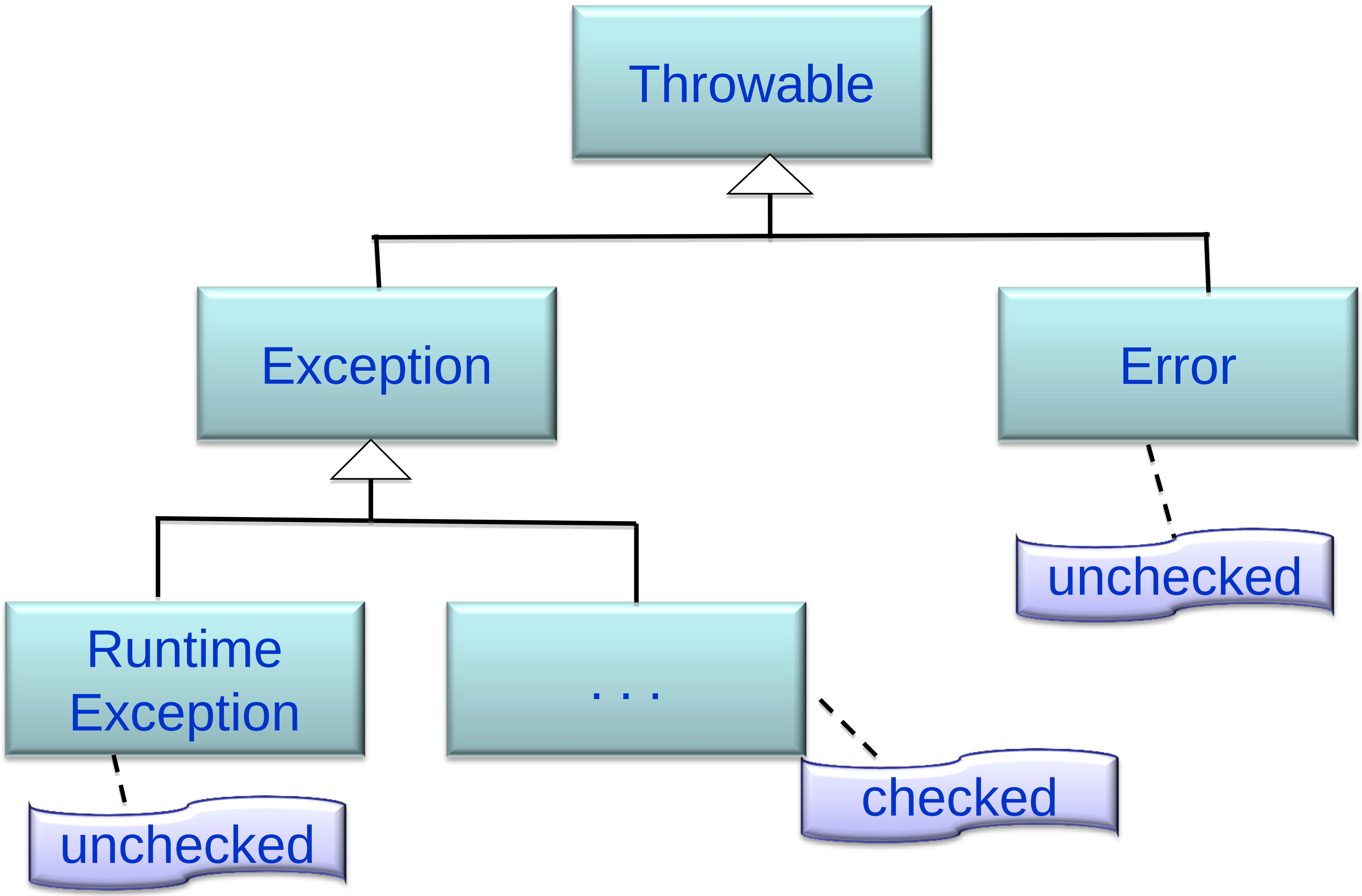
  - provide an exception handler (with a **try**/**catch**/**finally** block)
  - propagate the exception (whose type is declared within the **throws** clause) to the caller

- Unchecked exceptions

  May be handled, if desired

  Unhandled exceptions terminate the current execution thread

Exception class hierarchy



Exception handlers

The scope of an exception handler is denoted by a **try** block

Every **try** block is immediately followed by zero or more **catch** blocks, zero or one **finally** block, or both. At least one of **catch** blocks and **finally** block is required (otherwise, the **try** would be useless)

```
public int foo(int b) {
    try { if ( b > 3 ) {
            throw new Exception();
          }
        } catch (Exception e) { b++; }
          finally { b++; }
    return b;
}
```

Exception handlers: catch blocks

`catch` blocks can be exception-specific:

```
catch (ExceptionType name) { /* handler */ }
```

- Targets exceptions whose type conforms to `ExceptionType`
- `ExceptionType` must be a descendant of `Throwable`
- `name` behaves as a local variable inside the handler block
- A `catch` block of type `T` cannot follow a `catch` block of type `S` if `T` ≤ `S` (otherwise the `T`-type block would be shadowed)

Multi `catch` blocks (introduced in Java 7):

```
catch (ET1 | ET2 | ET3 name) { /* handler */ }
```

- Targets exceptions whose type conforms to `ET1`, `ET2`, or `ET3`
- `ET1`, `ET2`, and `ET3` cannot be related by subclassing
- `name` behaves as a constant (`final`) inside the handler block

Exception handlers: catch/finally blocks

When an exception of type **`T`** is thrown within a **`try`** block:

- control is transferred to the first (in textual order) **`catch`** block whose type **`T`** conforms to, if one exists
- then, the control is then transferred to the **`finally`** block (if it exists)
- finally, execution continues after the **`try`** block

When no conforming **`catch`** exists or an exception is re-thrown inside the handler:

- After executing the **`finally`** block, the exception propagates to the next available enclosing handler

When a **`try`** block terminates without exceptions:

- the control is transferred to the **`finally`** block (if it exists)
- then, execution continues after the **`try`** block

Exception handlers: catch/finally blocks

A **finally** block is always executed after the **try** block even if no exceptions are thrown

- Typically used to free resources

```
// foo() returns 2 (!)
public int foo() {
   try { return 1; } finally { return 2; }
}
```

A control-flow breaking instruction (**return**, **break**, **continue**) inside a **finally** block terminates the propagation of exceptions.

```
// foo() returns 2 and propagates no exception
public int foo() {
  try {throw new Exception();} finally {return 2;}
}
```

Exception handlers

A **catch** block may contain other **try** blocks

From within a **catch** block an exception can be re-thrown:

```
catch (Exception e) { if (...) {throw e;} ...}
```

Exceptions that propagate to the **main** method without being handled force termination of the program (typically, showing a trace of the call stack).

Catch, handle, and re-throw: example

A method

```
int readNum(String fn, int n)
```

tries to read an **`n`**-digit integer from file with name **`fn`**.

Exceptions handle things that may go wrong:

- a file with name **`s`** doesn't exist
- the file cannot be opened
- the file doesn't encode an integer
- the integer has fewer than **`n`** digits

Catch, handle, and re-throw: example

```
public int readNum(String fn, int n)
   throws TooFewDigitsException, FileNotFoundException,
          IOException {
   int res;  BufferedReader br = null;
   try {
       br = new BufferedReader(new FileReader(fn));
       String str = br.readLine();
       if (str.length < n)
         throw new TooFewDigitsException(str.length);
       res = Integer.parseInt(str);
   }
   catch (FileNotFoundException e) { throw e; }
   catch (IOException e) { throw e; }
   catch (NumberFormatException e) { res = 0; }
   finally { if (br != null) br.close(); }
   return res;    }
```

Catch, handle, and re-throw: example

Here's how a client may use **readNum**:

```
int readInt;
String aFileName;
try {
   readInt = n.readNum(aFileName, 5);
}
catch (TooFewDigitsException e) {
   try { readInt = n.readNum(FileName, e.numRead); }
   catch (Exception e) {System.out.println("Give up!");}
}
catch (Exception e) { System.out.println("IO error"); }
```

Try with resources

Starting with Java 7, a **try** may also list some resources that are automatically closed after the block terminates (as normally done explicitly within a **finally** block).

```
try (
  FileOutputStream out = new FileOutputStream(“o.txt");
  FileInputStream in = new FileInputStream(“i.txt");
) {'
    // code that uses ‘out’ and ‘in’
} catch (IOException e) { /* Couldn’t open files */ }
```

catch and **finally** are completely optional in try-with-resources blocks (but checked exceptions must still be caught or propagated).

A class must implement interface **java.lang.AutoCloseable** to be usable in a try-with-resources block.

- Basically, it needs a **close()** method

Checked vs. unchecked exceptions

Checked exceptions are quite unique to Java

- C++ and C#, in particular, have only the equivalent of unchecked exceptions

Which type of exception should you use in your Java programs?

Java orthodoxy: checked exceptions should be the norm

Rationale for preferring checked exceptions:

- exceptions usually carry information the client of a class should be informed about
- a method throwing unchecked exceptions is similar to a method with undocumented behavior
- clients are generally unprepared to deal with unexpected exceptions

Checked vs. unchecked exceptions

Disadvantages of using checked exceptions extensively:

- lots of exception handling code to write
  - lazy programmer's shortcut: empty `catch` blocks
- many `catch` blocks pollute code and decrease readability
- complex unwinding of the call stack to decide which exceptions to propagate and which to handle
- new exceptions change the interface of methods

Checked vs. unchecked exceptions

How to strike a balance:

- As a norm, checked exceptions should replace error codes when the client should check the return code
- Use a checked exception if the caller can do something sensible with the exception
  - useless with fatal errors whose causes are outside of the client’s influence
- Document the usage of unchecked exceptions
- Don’t use exceptions (checked or unchecked) when you should use assertions (contracts)
  - see examples in C# slides of this class

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Genericity in Java

Generics

Java's genericity mechanism, available since Java 5.0

Most common use:

- Use (and implement) generic type-safe containers

```
ArrayList<String> safeBox = new ArrayList<String>();
```

- Compile-time type-checking is enforced

More sophisticated uses:

- Custom generic classes and methods
- Bounded genericity (also called constrained genericity)

```
public <T extends Interface1 & Interface2> T test(T x)
```

Generic classes

A generic class is a class parameterized w.r.t. one or more generic types.

```
public class Cell<T> {
    private T val;
    public T getVal() { return val; }
    public void setVal(T v) { val = v; }
}
```

To instantiate a generic class we must provide an actual type for the generic parameters.

```
Cell<String> c = new Cell<String>();
```

Generic classes

The generic parameters of a generic class may constrain the valid actual types.

```
public class Cell<T extends S> { ... }
```

The following is valid only if **X** is a subtype of **S**:

```
Cell<X> c = new Cell<X>();
```

The constrains may involve multiple types.

```
public class C<T extends String & Iterable>
```

The following is valid only if **Y** is a subtype of both **String** and **Iterable**:

```
C<Y> c = new C<Y>();
```

Genericity before generics

Before generics were available, using class **Object** was the way to achieve generic implementations.

```
public class OldCell {
    private Object val;
    public Object getVal() { return val; }
    public void setVal(Object v) { val = v; }}
```

Requires explicit castings, with major problems:

- verbose code
- no compile-time checks

```
OldCell c = new OldCell();
c.setVal("A string");  // upcasting
String s = (String) c.getVal(); // downcasting
Car c = (Car) c.getVal(); // runtime error
```

Diamond operators and raw types

When creating an instance of a generic class, the compiler is often able to infer the generic type from the context. In such cases, we can use the diamond operator.

```
Cell<String> c = new Cell<>();
```

is equivalent to:

```
Cell<String> c = new Cell<String>();
```

Generic classes can be instantiated as raw types, without providing any generic parameter. Raw types correspond to the old type-unsafe generic classes:

```
Cell c = new Cell();
c.setVal(12); // warning of unsafe behavior
Cell<String> c = new Cell();
    // not equivalent to new Cell<>()!
```

Generics: features and limitations

Generic classes are translated into ordinary classes by the compiler:

- Process called “type erasure”
- The generic type is replaced by **`Object`**
- Casts are added as needed, after checking that they are type-safe

Limitations of type erasure:

- Can’t instantiate generic parameter with primitive types
  - but can use wrapper classes
- At runtime you cannot tell the difference between **`ArrayList<Integer>`** and **`ArrayList<String>`**
- Exception classes cannot be generic classes
- Can’t create objects of a generic type
  - but can assign the value **`null`** to a variable of generic type
- Arrays with elements of a generic type parameter cannot be created
- A static member cannot reference a generic type parameter

Generics and inheritance

Let S be a subtype of T (i.e. S ≤ T)

There is no inheritance relation between:

SomeGenericClass<S> and **SomeGenericClass<T>**

In particular: the former is not a subtype of the latter

However, let **AClass** be a non-generic type:

- **T<AClass>** is a subtype of **T**
    - **T** denotes the raw type derived from the generic class **T**
- **S<AClass>** is a subtype of **T<AClass>**

Why subtyping with generics is tricky

Consider a method of class **F**:

```
public static void foo(LinkedList<Vehicle> x){
    // add a Truck to the end of list 'x'
    x.add(new Truck());
}
```

If **LinkedList<Car>** were a subtype of **LinkedList<Vehicle>**, this would be valid code:

```
LinkedList<Vehicle> cars = new LinkedList<Car>();
cars.add(new Car());
F.foo(cars);
```

But now a **LinkedList<Car>** would contain a **Truck**, which is not a **Car**!

Wildcards

Give some polymorphic features to generics

Unbounded wildcards: **`Collection<?>`**

- "**`Collection`** of unkwnown(s)"
- It is a super-type of **`Collection<T>`**, for any class T
  - A method can read elements from a wildcard collection argument
  - Can assign elements of the collection to references of type **`Object`**
  - Cannot add new elements to the collection (see previous example)
  - But it can add new **`null`** entries
    - because **`null`** is a subtype of every other type

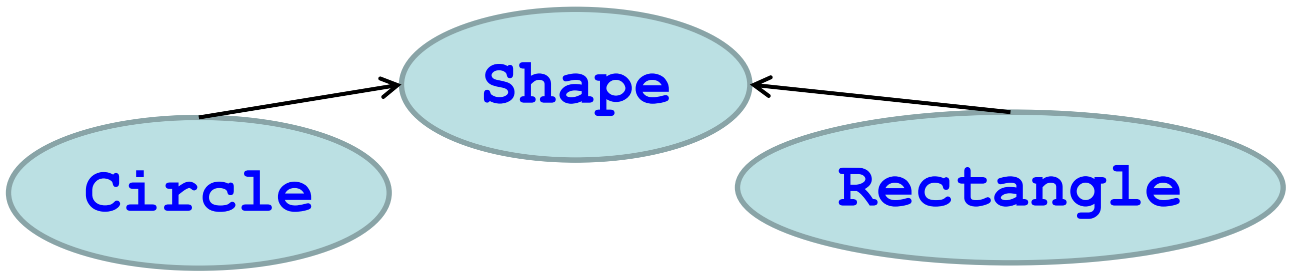
Bounded wildcards

Bounded wildcards with upper bound:
`Collection<? extends X>`

- It is a super-type of `Collection<T>`, for any subclass `T` of `X`
  - A method can read elements from the wildcard collection argument
  - Can assign elements of the collection to references of type `X`
  - Cannot add new elements to the collection
  - But it can add new null entries
    - because `null` is a subtype of every other type

Upper-bounded wildcards: example

Consider the following hierarchy of classes:



What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

- **drawShapes(List<Shape> shapes)**
  - this doesn't work on a **List<Circle>**, which is not a subtype of **List<Shape>**
- **drawShapes(List<? extends Shape> shapes)**
  - this works on **List<Shape>**, **List<Circle>**, and **List<Rectangle>**, but doesn't work on **List<Object>** (correctly, as drawing is not defined for something that may not be a **Shape**)

Bounded wildcards

Bounded wildcards with lower bound:
`Collection<? super X>`

- It is a super-type of **`Collection<T>`**, for any superclass **`T`** of **`X`**
  - A method can add elements to the collection (i.e., through the wildcard collection argument)
  - Cannot assign elements of the collection to references of type **`X`**
  - But it can read elements and assign them to reference of type **`Object`**
    - because **`Object`** is a supertype of every other type

Lower bounds are often used for write-only resources such as log streams.

Lower-bounded wildcards

Consider a class for a list, including a sort method:

```
class MySortedList <T> implements List
{   ...
    void sort(Comparator <T> cmp) { ... }
    ...
}
```

- `MySortedList<String> sl = new MySortedList<>();`
  `Comparator<String> mc = ... ;`
  `Comparator<Object> oc = ... ;`
- Valid call: `sl.sort(mc);`
- Invalid call: `sl.sort(oc);`
  - `Comparator<Object>` is incompatible with `Comparator<String>`
- Solution: use a lower-bounded wildcard in `sort`'s signature
  `void sort(Comparator <? super T> cmp)`

Generic methods

They are useful where wildcards fall short:

adding elements to a generic collection

Example: defining a method that assigns the elements in an array to a generic collection

```
static void a2c(Object[] a, Collection<?> c) {
  for (Object o : a) { c.add(o); /* Error */ } }
```

- We will know whether the type of `o`'s elements is compatible with the type of `c`'s elements only at runtime

Generic methods

Example: defining a method that assigns the elements in an array to a generic collection

Generic methods come to the rescue (notice the position of the generic parameter):

```
static <G> void a2c(G[] a, Collection<G> c) {
  for (G o : a) { c.add(o); /* OK */ } }
```

This is how client use the generic method.

```
String[] arr = {“Hello”, “world”, “!”};
ArrayList<Object> lst = new ArrayList<>();
a2c(arr, lst);
```

The actual generic parameter is inferred from context.

Collections

A classic example of separating interface from implementation

Some useful library interfaces from `java.util`:

- `Collection<E>`
  - `boolean add(E el)`
    - returns whether the collection actually changed
  - `void clear()`
    - remove all elements in the collection
  - `Iterator<E> iterator()`
    - returns an iterator over the collection
- `Iterator<E>`
  - `E next()`
  - `void remove()`
    - removes the last element returned by the iterator

Collections: some implementations

- ArrayList: indexed, dynamically growing
- LinkedList: ordered, efficient insertion and removal
- HashSet: unordered, rejects duplicates
- TreeSet: ordered, rejects duplicates
- HashMap: key/value associations
- TreeMap: key/value associations, sorted keys

Java collections framework

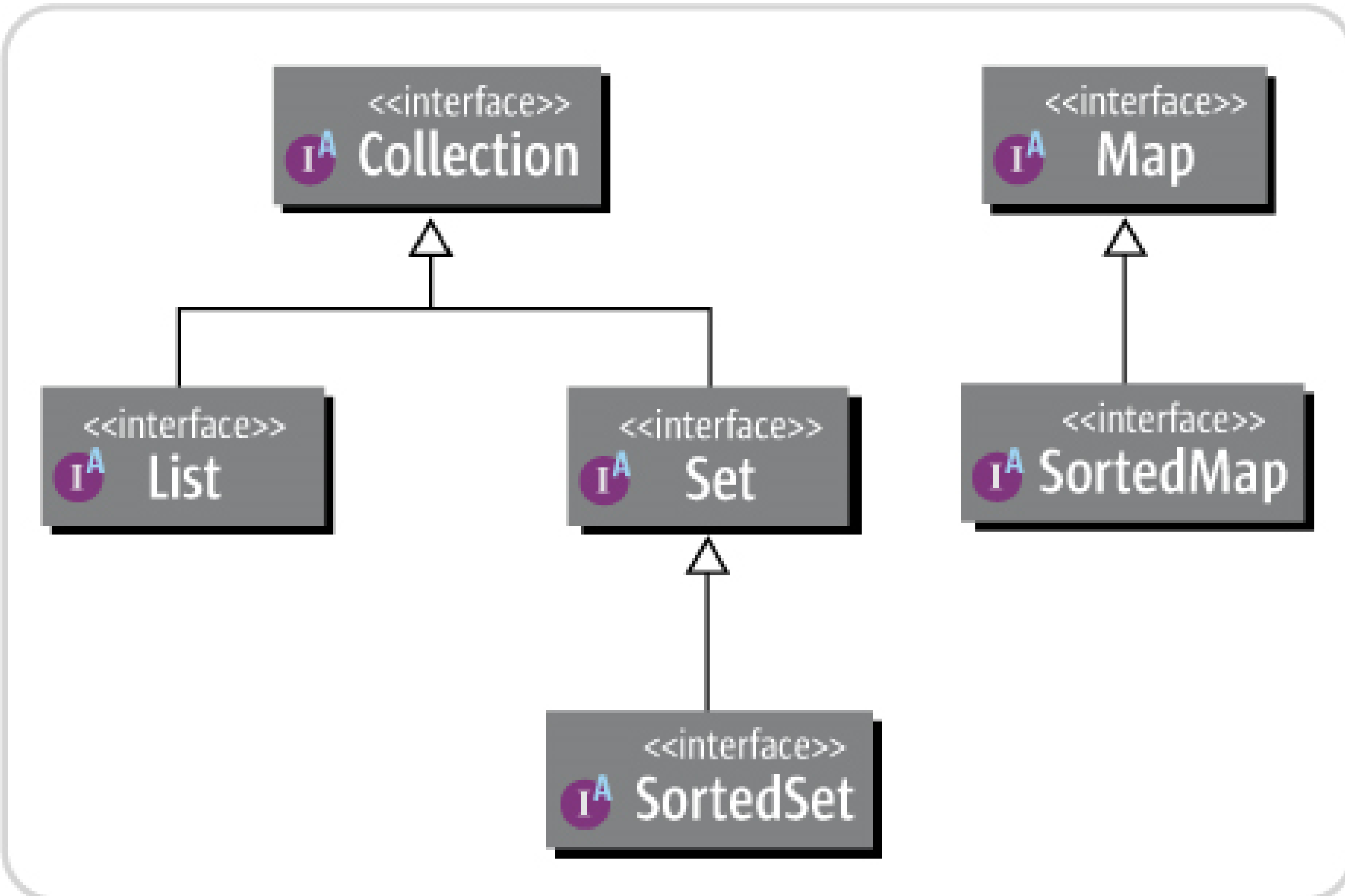


Figure 1 Collections Framework major interfaces

Java collections framework

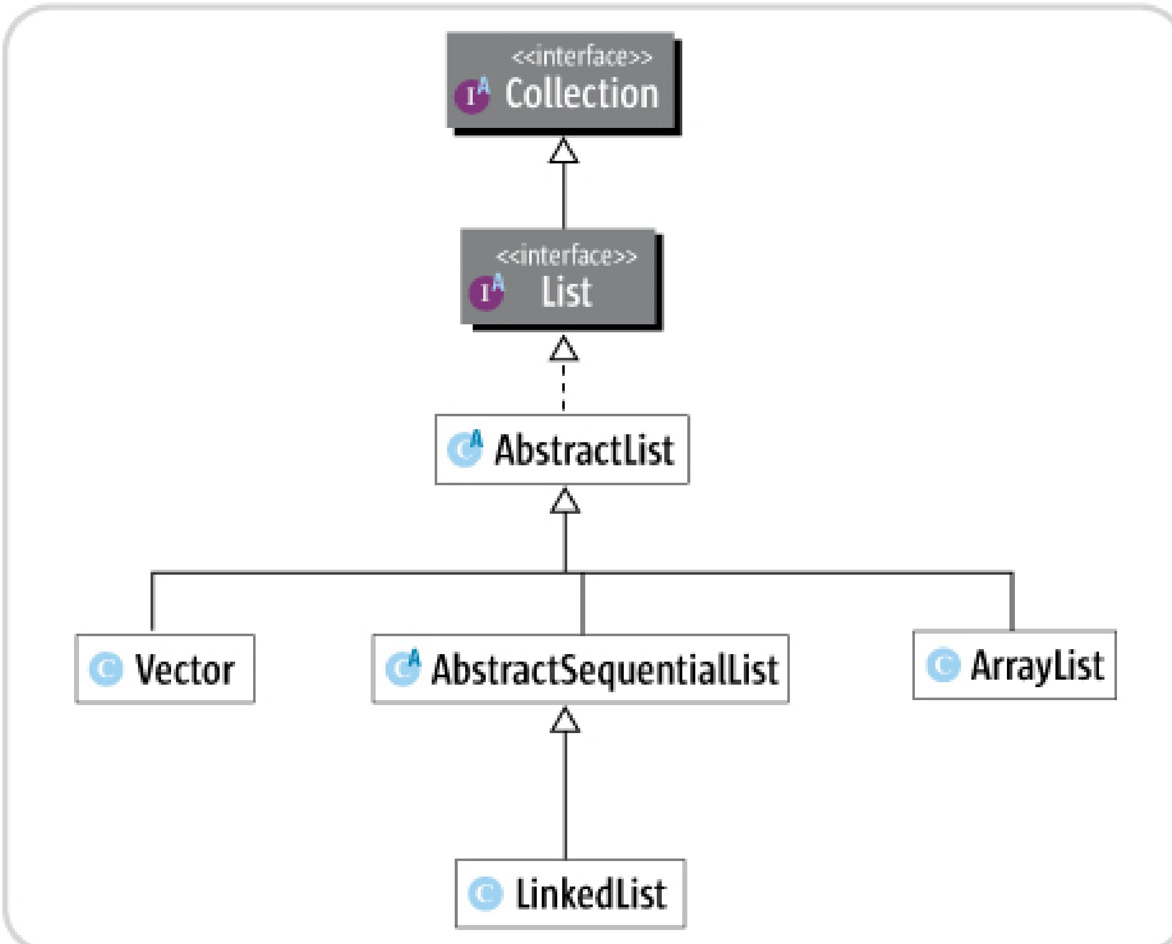


Figure 3 List category

Java collections framework

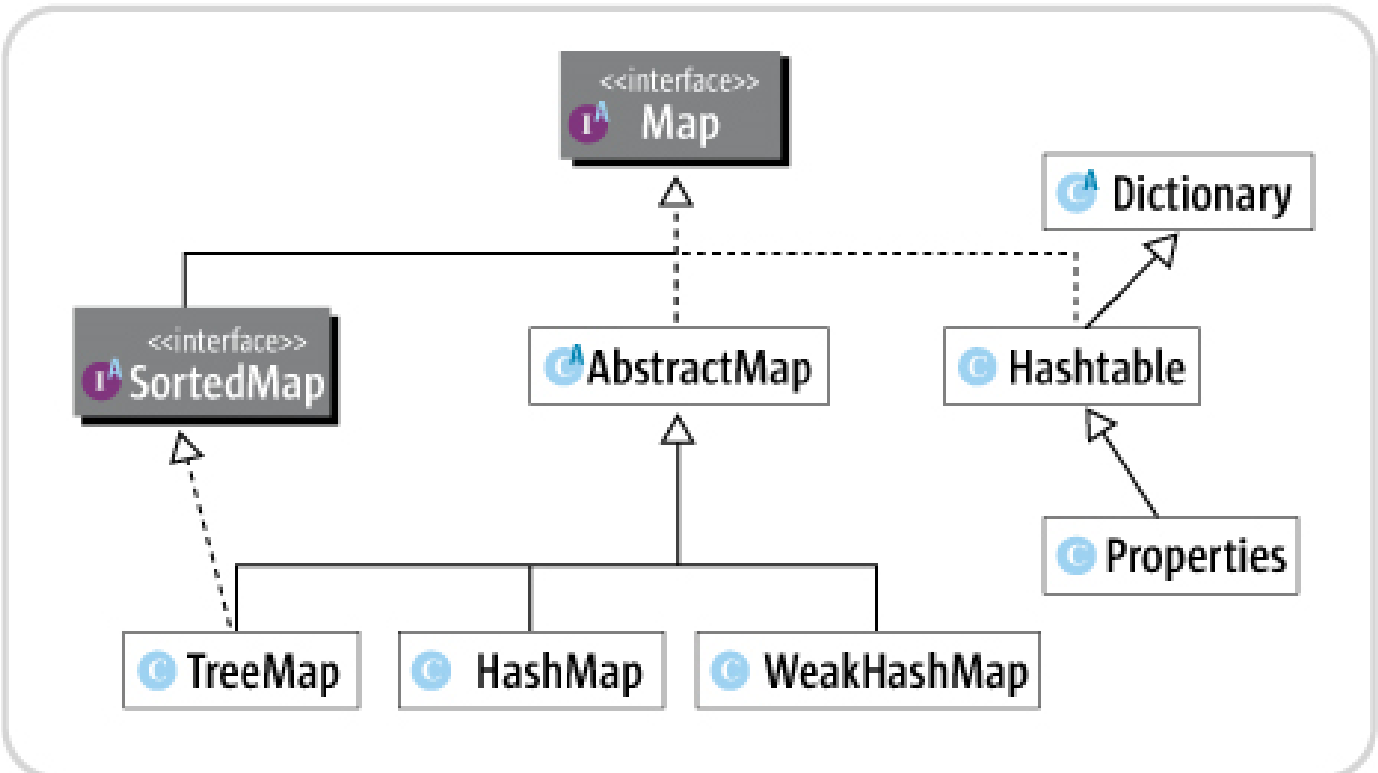


Figure 4 Map category

Java collections framework

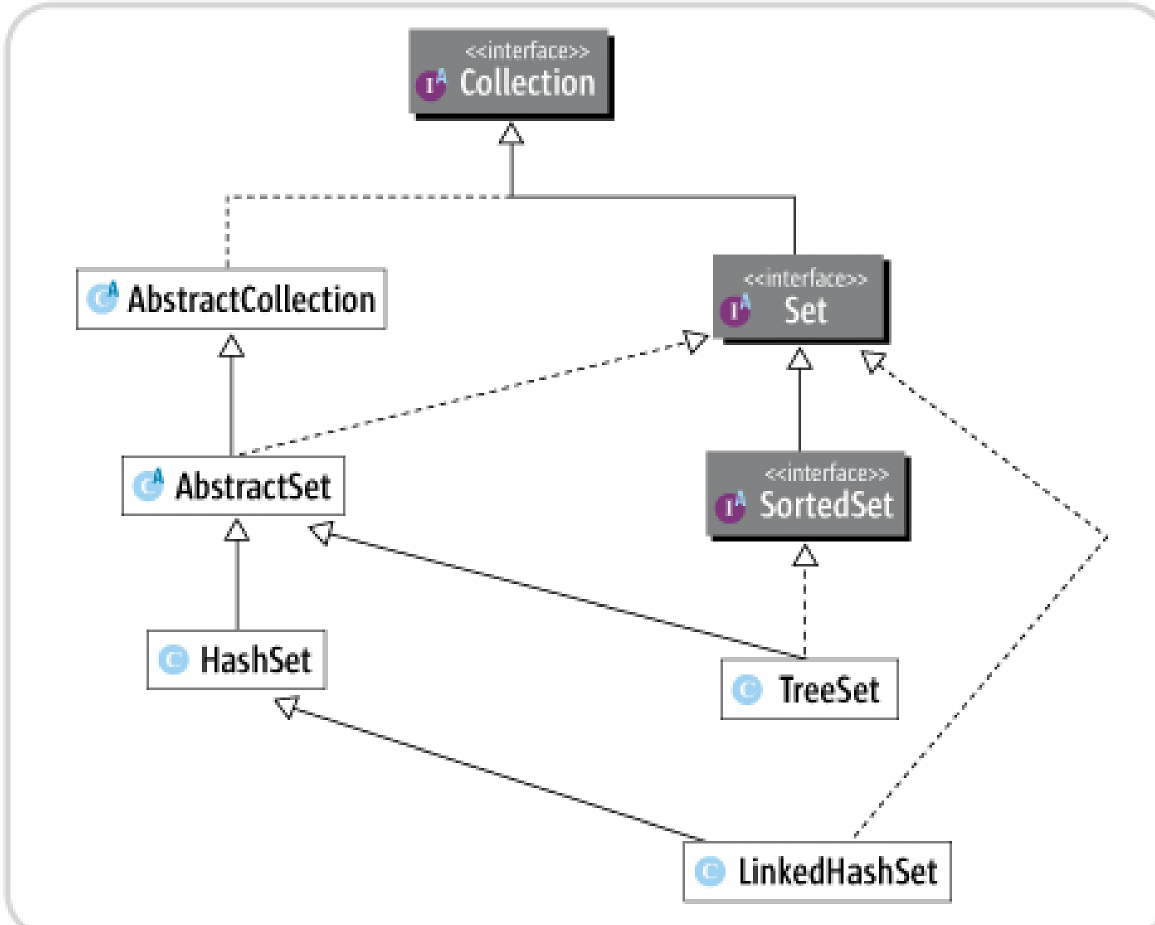


Figure 2 Set category