# Cast expression (*S*) *e*: when is it type correct?

## Simplified static semantics where *e* has type *T*

- all casts among primitive types allowed, except for **`boolean`** that can only be cast to itself (not so useful!)
  Remark: numeric conversions with cast are efficient, but inaccurate; for accuracy better using the conversion methods of `Math`
- boxing +(optionally) widening reference conversion
  example: *T*=**`int`**, *S*=`Number`
- unboxing +(optionally) widening primitive conversion
  example: *T*=`Integer`, *S*=**`long`**
- narrowing reference conversion +(optionally) unboxing
  example: *T*=`Number`, *S*=**`int`**
- a reference type *T* can be cast to another reference type *S* if there can exist $T'$ s.t. $T' \leq S$ and $T' \leq T$
  - *T* and *S* are both classes, and $T \leq S$ or $S \leq T$;
  - *S* and *T* are both interfaces;
  - ...

# Cast expression ( $S$ ) $e$: what is its behavior?

## Simplified dynamic semantics where $e$ has type $T$

There are 3 most common cases:

1. widening/narrowing primitive conversion
   $T$ and $S$ different primitive numeric types
   action: the value of type $T$ is converted into a value of type $S$
2. widening reference conversion
   $T$ and $S$ reference types, $T \leq S$
   action: no action is performed
3. narrowing reference conversion
   $T$ and $S$ reference types, $T \not\leq S$
   action:
   (1) the dynamic type $T'$ of $e$ is tested to be a subtype of $S$
   (2) if the test succeeds no other action is performed
   otherwise an exception of type ClassCastException is thrown

Remarks:

- the dynamic type $T'$ of $e$ is the type of the value of $e$
- $T'$ is usually different from $T$, but always $T' \leq T$

# Example of use of cast expressions

The most useful cast is widening reference conversion to select overloaded methods

## Example

```
// String contains both valueOf(char[]) and valueOf(Object)
char[] c = {'a','b','c'};
String.valueOf(c);                  // valueOf(char[]) is called
String.valueOf((Object) c);         // valueOf(Object) is called
String.valueOf((Object) null); // valueOf(Object) is called
```

Remark: the type of `null` is subtype of any reference type

Since Java 16 casts for narrowing reference conversion are seldom needed (see next slides)

# Java `instanceof` (new version since Java 16)

## Syntax

- $e$ **`instanceof`** $T$ $x$
- $T$ $x$ pattern with $T$ type and $x$ variable

## Dynamic semantics

If the value $v$ of $e$ is different from **`null`** and has type $T' \leq T$,
then **`true`** is returned and variable $x$ is declared of type $T$ and initialized with $v$
otherwise **`false`** is returned and no variable is declared

## Static semantics

- the static type of $e$ must be a reference type
- the cast `(`$T$`)`$e$ must be type correct

# Java **instanceof** (new version since Java 16)

The use of **instanceof** is useful in conjunction with the conditional statement

## Example

```
public boolean equals(Object otherTimer) {
  if (otherTimer instanceof Timer t) // t is a local variable in the then-branch
     return time == t.getTime();
  return false;
}
```

Remark: before Java 16 no variable could be used in **instanceof**

# Generic types: motivations

## Example of bad class definition

```
import java.awt.Color;
public class Pair { // bad definition!
    private final Object fst;
    private final Object snd;

    public Pair(Object fst, Object snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public Object getFst() { return fst; }
    public Object getSnd() { return snd; }
}
...
Pair p = new Pair("a string", Color.RED);
String s = (String) p.getFst(); // reference narrowing needed!
Color c = (Color) p.getSnd();   // reference narrowing needed!
```

Fix: use parametric polymorphism instead of subtyping polymorphism

# Generic types: motivations

## Better solution: `Pair` is a generic class

```
import java.awt.Color;
public class Pair<T1,T2> { // T1, T2 type parameters
    private final T1 fst;
    private final T2 snd;
    public Pair(T1 fst, T2 snd) { this.fst = fst; this.snd = snd; }
    public T1 getFst() { return fst; }
    public T2 getSnd() { return snd; }
}
...
Pair<String,Color> p = new Pair<String,Color>("a string", Color.RED);
String s = p.getFst();
Color c = p.getSnd();
Pair<Color,String> p2 = new Pair<Color,String>(Color.RED,"a string");
p=p2; // type error!!
```

## Diamond notation

with **new**, type arguments can be inferred in many cases

```
Pair<String,Color> p = new Pair<String,Color>("a string", Color.RED);
```

can be rewritten in this simpler way:

```
Pair<String,Color> p = new Pair<>("a string", Color.RED);
```

# Generic types

- classes/interfaces with type parameters are called generic
- `Pair` is a generic class with two type parameters `T1` and `T2`
- `Pair<String,Color>`/`Pair<Color,String>` are parameterized types
- `String`/`Color` are the argument types in `Pair<String,Color>`
- parameters `T1` and `T2` are instantiated with `String` and `Color`
- type arguments can only be reference types
- examples of incorrect uses of `Pair`:

```
Pair<int,String>
Pair<String>
Pair<String,String,String>
```

- generic classes support parametric polymorphism

## Example with `p` of type `Pair<String,Color>`

- `p.getFst()` has static type `String`
- `p.getSnd()` has static type `Color`

# Use case for generic types

## Basic Java API interfaces for object containers

```
package java.util;

public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean add(E e);
  ...
}

public interface List<E> extends Collection<E> {
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);
  ...
}
```

The parameter `E` corresponds to the type of the elements of the object containers

# Use case for generic types

## Basic Java API classes for object containers

```
package java.util;

public abstract class AbstractCollection<E> extends Object implements
    Collection<E>  {...}

public abstract class AbstractList<E> extends AbstractCollection<E>
    implements List<E> {...}

// implementation with arrays
public class ArrayList<E> extends AbstractList<E> {
  public ArrayList() {...}
  ...
}

// implementation with doubly-linked lists
public class LinkedList<E> extends AbstractList<E> {
  public LinkedList() {...}
  ...
}
```

# Problem: iterating over object containers

## A standard programming problem

Iterating over the elements of an object container to perform some operation
Simple examples:

- find the first elements in a list satisfying a given property
- compute the sum of all elements in a list/set
- compute the max element in a list/set

## Is there a general way to solve this problem?

Desiderata:

- efficient time complexity (ideally linear)
- same code for different object containers
  - different implementation of the same Abstract Data Type: e.g. array lists, linked lists
  - differen Abstract Data Types: lists, sets, maps, . . .

# Iteration on object containers

## A first unsatisfactory attempt

```
// requirement: works on both ArrayList and LinkedList
public static int search(int e, List<Integer> ls) {
    for (int i = 0; i < ls.size(); i++)
        if (ls.get(i) == e) // element at index i, implicit unboxing
            return i;
    return -1;
}
```

## Example with array lists

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 1; i <= 6; i++)
   list.add(i); // appends i to the end of list
assert search(5, list) == 4; // linear time complexity
```

# Iteration on object containers

## A first unsatisfactory attempt

```java
// requirement: works on both ArrayList and LinkedList
public static int search(int e, List<Integer> ls) {
    for (int i = 0; i < ls.size(); i++)
        if (ls.get(i) == e) // element at index i, implicit unboxing
            return i;
    return -1;
}
```

## Example with linked lists

```java
List<Integer> list = new LinkedList<Integer>();
for (int i = 1; i <= 6; i++)
   list.add(i); // appends i to the end of list
assert search(5, list) == 4; // quadratic time complexity!
```

# Iterator design pattern

## Problems

- `List<E>` interface is not abstract enough
- no separation between iteration and element processing
- not as efficient as it could be
- code specific for lists, not general enough

## Solution

Implement external iterators with the iterator design pattern

## What is a design pattern?

A general and principled solution to some kind of programming problem

## External vs internal iterators

- iterator pattern: purely o.-o. approach based on external iterators
- internal iterators: another pattern inspired by functional programming

# Iterator design pattern

## Iterable and iterator objects

- Iterable objects: the objects for which it is possible to iterate over their elements with an iterator
- Iterator objects: the objects used to iterate over iterable objects
- with objects of type `Iterable` it is possible to use the enhanced-for to iterate over their elements
- more iterators may be active at the same time on the same iterable object

# Iterator design pattern

## `java.lang.Iterable<E>` and `java.util.Iterator<E>`

```
public interface Iterable<E> {
    Iterator<E> iterator(); // factory method
}

public interface Iterator<E> {
    boolean hasNext();      // true iff the iteration has more elements
    E next();               // the next element, may throw NoSuchElementException
    ...
}
```

# Iterator design pattern

## Implementation of `search` with the iterator pattern

```
public static int search(int e, List<Integer> ls) {
    int res = 0;
    Iterator<Integer> it = ls.iterator(); // returns a new iterator on ls
    while (it.hasNext()) {
        if (it.next() == e) return res;
        res++;
    }
    return -1;
}

// more concise version with the enhanced for (for-each)
public static int search(int e, List<Integer> ls) {
    int res = 0;
    for (int el : ls) { // for-each with iterable objects or arrays
        if (el == e) return res;
        res++;
    }
    return -1;
}
```

Remark: `List<E>` implements `Iterable<E>`

# Iterator design pattern

## Another example

Sum all the integers contained in a collection of integers

## Solution

```
public static int sumAll(Collection<Integer> col) {
    int res = 0;
    Iterator<Integer> it = col.iterator();
    while (it.hasNext())
        res += it.next();
    return res;
}
// more concise version with the enhanced for (for-each)
public static int sumAll(Collection<Integer> col) {
    int res = 0;
    for (int i : col)
        res += i;
    return res;
}
```

Remark: `Collection<E>` implements `Iterable<E>`

# Iterator design pattern

## Demo

```
public class ArrayIterator implements Iterator<Integer> {
    private final int[] array;
    private int nextIndex;

    public ArrayIterator(int[] array) {this.array = requireNonNull(array);}

    public boolean hasNext() { return nextIndex < array.length; }

    public Integer next() {
        if(!hasNext())
            throw new NoSuchElementException();
        return array[nextIndex++];
    }
    public static void main(String[] args) {
         int[] a = new int[]{1, 2, 3};
         Iterator<Integer> it = new ArrayIterator(a);
         Iterator<Integer> it2 = new ArrayIterator(a);
         var el = 1;
         while (it.hasNext()) {
             assert it.next() == el;
             el++;
         }
         assert !it.hasNext() && it2.hasNext();
     }
}
```