## Defensive Programming

*Defensive programming* refers to programming practices that help prevent bugs.



## Defensive Coding Practices

- always use {} for loop and conditional bodies

```
if ( x < 0 )
  x = -x;
```

legal, but easy to add another line and get...

```
if ( x < 0 )
  System.out.println("x: "+x);
  x = -x;
```

valid code, but incorrect because x = -x has been kicked out of the 'if' body

```
if ( x < 0 ) {
  x = -x;
}
```

recommended solution



## Defensive Coding Practices

- don't ignore return values
  - especially when they indicate success or failure of the operation

```
// get the substring before ,
String before =
  str.substring(0,str.indexOf(','));
```

crashes if 'str' doesn't contain ,

```
int index = str.indexOf(',');
if ( index == -1 ) { … }   // handle error
String before = str.substring(0,index);
```

if it is valid for 'str' not to contain u

```
int index = str.indexOf(',');
assert index > -1;
String before = str.substring(0,index);
```

if 'str' is known to contain 'src'



## Defensive Coding Practices

- don't rely on default values
  - initialize all variables when they are declared – including slots of arrays if not partially full
  - explicitly set settings when things are created
- don't rely on "pass through" behavior in conditionals – cover all alternatives, even if expected to be impossible

```
int a = 0;
if ( b > 100 ) { a = 2; }
else if ( b > 10 ) { a = 1; }
```

  - what if b ≤ 10?
    - generate error if the case should be impossible
    - include final `else` to set `a = 0` otherwise
  - (this is not about "do nothing" alternatives)

```
int a;
if ( b > 100 ) { a = 2; }
else if ( b > 10 ) { a = 1; }
else { a = 0; }

int a = 0;  // for compiler
assert b > 10;
if ( b > 100 ) { a = 2; }
else if ( b > 10 ) { a = 1; }
```

- don't trust outside data
  - e.g. check that parameter values are legal

## Defensive Coding Practices

Note the kinds of errors you often make and consider ways to prevent them.

- e.g. always use `for` loops instead of `while` loops
  - `for` loops have an obvious place for the update step
  - `for` loops facilitate loop variables that are local to the loop, and initialize them before the loop begins
    - can help avoid problems with forgetting to initialize, especially in the inner loop in a set of nested loops



## Defensive Coding Practices

```
for ( int i = 0 ; i < 10 ; i++ ) {
  …
}
```

```
int i = 0;
while ( i < 10 ) {
  …
  i++;
}
// i is still in scope
```



## Defensive Coding Practices

```
for ( int i = 0 ; i < 10 ; i++ ) {
  for ( int j = i+1 ; j < 10 ; j++ ) {
  …
  }
}
```

int i = 0, j = i; is a mistake

i++; j++; is a mistake

```
int i = 0;
while ( i < 10 ) {
  int j = i;
  while ( j < 10 ) {
    …
    j++;
  }
  // j is still in scope
  i++;
}
// i is still in scope
```



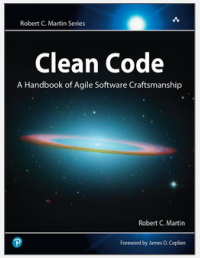
## Defensive Coding Practices

Note the kinds of errors you often make and consider ways to prevent them.

- e.g. write the literal first in an equality comparison
  - `if ( true == b ) { … }`

# Naming Practices

- good naming practices reduce confusion and potential bugs




[2008] note: some of Martin's clean code principles are controversial and viewed as outdated and some go against language conventions or traditions, but it is still valuable to understand the principles and the intent behind them

Robert "Uncle Bob" Martin
American software engineer, 1952-
known for SOLID principles (OO design principles), Agile Manifesto (software development methodology)

Uncle Bob picture credit:
By Angelacleancoder - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=88628927



```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for ( int[] x : theList ) {
    if ( x[0] == 4 ) {
      list1.add(x);
    }
  }
  return list1;
}
```

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for ( int[] cell : gameboard ) {
    if ( cell[STATUS_VALUE] == FLAGGED ) {
      flaggedCells.add(cell);
    }
  }
  return flaggedCells;
}
```

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for ( Cell cell : gameboard ) {
    if ( cell.isFlagged() ) {
      flaggedCells.add(cell);
    }
  }
  return flaggedCells;
}
```

# Clean Code – Names Should Reveal Intent

- what is in the list being iterated through?
- what is the significance of the 0th subscript of an item (`int[]`) in that list?
- what is the significance of the value 4?
- how would you use the list being returned?
- what does this function do?

 Robert Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* 

# Clean Code – Comments

Martin's clean code principles prefer writing code that doesn't need comments to be understandable – one reason is that comments can become out-of-date as code is changed.

This can go overboard – long names are unwieldy, comments aren't inherently bad – but the idea of choosing names that are understandable on their own is a good goal.

According to the clean code principles for comments, which of the following is preferred?

```
public class StackControlHeader {
  private int shgrow_;   // number of times stack has grown
  private int shaseg_;   // size of increments to stack
  private int shhwm_;    // high water mark of stack
  private int shsize_;   // current size of stack (all segment
s)
}
```

```
public class StackControlHeader {
  private int numTimesGrown_;
  private int sizeOfInc_;
  private int highWaterMark_;
  private int currentSize_;
}
```

```
public class StackControlHeader {
  private int numTimesGrown_;  // number of times stack has gr
own
  private int sizeOfInc_;      // size of increments to stack
  private int highWaterMark_;  // high water mark of stack
  private int currentSize_;    // current size of stack (all s
egments)
}
```

- all of these are fine
- none of these - there's a better alternative



# Code Smells

- avoid short variable names in large scopes
  - name length is a proxy for precision/amount of detail
  - single-letter names should be limited to local variables used in a limited context

```
for ( Cell cell : getLivingCells() ) {
  int count = countLivingNeighbors(cell);
  if ( count == 2 || count == 3) {
    ng.add(cell);
  }
}
```

Which of the following are examples of the "short variable names in large scopes" code smell? (Choose all that apply.)

- using cell instead of livingCell
- using count instead of numberOfLivingNeighbors
- **using ng instead of nextGeneration**

- `cell` and `count` are limited to a short loop – a short, general name is fine because the detail is clear from the context (`cell` is a living cell and `count` is the number of living neighbors); a longer name is tedious and hinders readability
- `ng` has a larger scope (so large that its declaration and initialization aren't visible), so it needs a longer, more descriptive name in order to be understood without its context – a short name is cryptic

# Code Smells

- avoid long function names in large scopes
  - length is a proxy for precision/amount of detail

```
public void computeNextGeneration () { ... }
private boolean isCellAlive ( Cell c ) { ... }
private int countLivingNeighbors ( Cell c ) { ... }
```

Which of the following are examples of the "long function scopes" code smell? (Choose all that apply.)

- using isCellAlive instead of living
- using countLivingNeighbors instead of count
- using computeNextGeneration instead of computeNextGen
- **using computeNextGeneration instead of step**

- public methods have a larger scope than private methods
- methods with a larger scope are likely called more often than those with a smaller scope and also are more likely to have a higher level of abstraction, so shorter, less detailed names are better – `step` instead of `computeNextGeneration`
- local methods are called less often and are more likely to be at a lower level of abstraction, so greater detail is appropriate and longer names can act as a form of documentation



# Questions

What is a good guideline to determine a short vs long name?

- it's less about the number of characters as such, and more about the level of detail in the name
- variables
  - short, general name is OK in a small scope since there's context to make it understandable
  - longer, more specific name in a large scope in order to be understood without the context
- methods
  - short, general name is OK in a large scope because those methods are more likely to be at a higher level of abstraction and to be called more often
  - longer, more specific name in a small scope is OK because likely called less often and long name is a form of documentation for a more specific task



# Clean Code – Reduce Vertical Scope

- reduce distance between declarations and use
  - declare as locally as possible
  - declare just before first use

"Reduce vertical scope" means which of the following is preferred?

```
protected void parseColumnHeader() {
  int headerColumns = table.getColumnCountInRow(1);
  for ( int col = 0 ; col < headerColumns ; col++ ) {
    String cell = table.getCellContents(col, 1);
    if ( cell.endsWith("?") ) {
      funcs.put(cell.substring(0, cell.length() - 1), col);
    } else {
      vars.put(cell, col);
    }
  }
}
```

```
protected void parseColumnHeader() {
  int headerColumns, col;
  String cell;

  headerColumns = table.getColumnCountInRow(1);
  for ( col = 0 ; col < headerColumns ; col++ ) {
    cell = table.getCellContents(col, 1);
    if ( cell.endsWith("?") ) {
      funcs.put(cell.substring(0, cell.length() - 1), col);
    } else {
      vars.put(cell, col);
    }
  }
}
```

- both are fine
- none of these - there's a better alternative



# Clean Code – Avoid Disinformation

"Avoid disinformation" means that which of the following is preferred?

- Account[] accountList
- **Account[] accounts**
- both are fine
- none of these - there's a better alternative

- "accountList" makes it sound like the variable is of type `List` (which then implies certain methods are applicable)
  - be careful when words have a broader or more generic usage in English but a more narrow technical meaning or connotation in Java – look for an alternative if the term is appropriate only in one sense

# Clean Code – Meaningful Distinctions

"Make meaningful distinctions" means which of the following is preferred?

- ○ public void move ( Position from, Position to )
- ○ public void move ( Position pos1, Position pos2 )
- ○ both are fine
- ○ none of these - there's a better alternative

- distinction here is between the two parameters
  - "from", "to" clarify the distinction
  - "pos1", "pos2" does not



# Clean Code – One Word Per Concept

- add (arithmetic) and add (to a collection) are different concepts
  - use different words
- there are multiple ways to add to a collection (at the end, at a particular position)
  - use different words
  - "append" is more descriptive for add-at-end

"Pick one word per concept" means that which of the following is preferred?

○
```
public int add ( int a, int b ) {
  return a+b;
}

public String add ( String s1, String s2 ) {
  return s1+s2;
}

public void add ( List<String> names, String player ) {
  names.add(player);
}
```

○
```
public int add ( int a, int b ) {
  return a+b;
}

public String add ( String s1, String s2 ) {
  return s1+s2;
}

public void append ( List<String> names, String player ) {
  names.add(player);
}
```

○
```
public int add ( int a, int b ) {
  return a+b;
}

public String concatenate ( String s1, String s2 ) {
  return s1+s2;
}

public void append ( List<String> names, String player ) {
  names.add(player);
}
```

- ○ all are fine
- ○ none of these - there's a better alternative



# Clean Code

- "Pick one word per concept" means which of the following is preferred?

```
public int add ( int a, int b ) { return a+b; }
public double add ( double a, double b ) { return a+b; }
```

```
public int addInt ( int a, int b ) { return a+b; }
public double addDouble ( double a, double b ) {
  return a+b;
}
```

  - use overloaded terms when the uses all match the same sense of the term and different terms for different senses